Cellular-Automata Based
Computational Modeling and Simulation
Of the Immune System

Department of Computer Science
University of Colorado at Colorado Springs
1420 Austin Bluffs Parkway
Colorado Springs CO 80918
Abstract

We have developed a software system called SIMISYS that models and simulates various aspects of the human immune system based on the computational framework of *cellular automata*. We are motivated by the goal of modeling participants in the immune system at the cell level, simulate their interactions and infer overall system behavior. We model tens of thousands of cells as exemplars of the significant players in the functioning of the immune system, and simulate normal and simple disease situations. We present the simulation while in progress with graphical illustration of the participating cells and appropriate graphs. SIMISYS 0.3, the current version of the software, is able to model and simulate the innate and adaptive components of the human immune system. The specific players of the immune system we model are the Macrophages, Dendritic cells, Neutrophils, Natural Killers, B Cells, T Helper Cells, Complement proteins and pathogenic bacteria.
1.0 INTRODUCTION

When a foreign substance or microbe (antigen) enters our body, the immune system acts to neutralize or destroy the foreign entity. The natural immune system provides a multilevel defense against the intruders through collective and coordinated response of approximately $10^{12}$ cells. The Immune System is a collection of molecules, cells and organs whose complex interactions and communications form an efficient system that is usually able to protect the individual from outside invaders. It is difficult to clearly differentiate the elements of the Immune System from the rest of the body. However, a traditional, if somewhat inadequate method, divides it into two functionally distinct parts: those elements that are Innate (Non-Adaptive), and those, which are acquired (Adaptive). These two parts are in no way isolated to perform their duties independently. There is a substantial amount of dialog between them. Innate immunity is characterized by its non-specificity, and is mediated by the actions of Phagocytes, Natural Killer (NK) Cells and Complement proteins. Adaptive immunity that is usually found only in vertebrates is mediated by B- and T-lymphocytes, clonally distributed and characterized by specificity and memory.

The main function of the immune system is to recognize all cells and molecules within the body and categorize them as self or non-self. Our body maintains a large number of immune cells, which circulate throughout the body. These immune cells are developed in bone marrow and travel to different part of the body through blood vessels. Different types of immune cells play different roles in the overall immune response. We usually visualize the immune system as a collection of independently operating cells, without any central control. But this is not true. Chemical signals are involved in the system to provide communication between these cells and they play an important role in the coordination of these diverse cells. In this study we have attempted to model several features of the innate and adaptive immune responses such as Self-nonself recognition, different phases of a macrophage’s life and their functions, attraction of neutrophils and Natural Killer cells to the region of attack and pathogen invasion and proliferation etc.

All immune cells are modeled as classes using object-oriented technology. Cellular interactions are modeled based on computational paradigm of cellular automata. A graphical user interface is also provided so that the user can vary parameters to model the immune cells and pathogen (bacteria) and run the simulation with a variety of inputs. The graphical display of the simulation is created using the SDL package providing visual images of what is going on in the simulation.

The organization of this paper is as follows. Section 2 provides a brief overview of the related research. Section 3 briefly describes the architecture of the software system designed and implemented. Section 4 provides details of the software developed. Here we give a brief overview of the systems approach we use to develop the model. The object-oriented class structure is also discussed in this Section. In Section 7 we analyze the simulation results and finally in Section 8, we conclude the paper by discussing the results and future directions for research.
2.0 RELATED RESEARCH

In spite of the enormous complexity of the immune system, several computational studies have focused on presenting a global and necessarily simplified understanding of the system and accompanying simulations. Three main approaches have been adopted by researchers:

1) Using ordinary differential equations to model various phenomena that take place in the immune system [1],
2) Using qualitative, i.e., non-numeric, information for modeling (e.g., Trelease 1996), and
3) Using distributed computation using cellular automata (e.g., Klienstein2000).

Ordinary differential equations (ODE) have been traditionally used to model complex systems such as the immune system in mammals where a large number of players of various kinds respond in different ways to the presence of various kinds of stimulants or chemicals. Researchers have also enumerated problems with pure ODE approaches (Klienstein 2000, Schatten 1999). These include a) The ODE approach assumes large populations of essentially identical entities, which is not the case with biological cells as each cell has a unique life history that defines its interaction with the environment, b) The ODE approach gives only average behavior of the system, and c) It is difficult to model non-linear behavior.

Cellular automata (Dewdney 1989, Toffoil 1987, von Newmann 1966, Wolfram 1986) are discrete dynamical systems whose behavior is completely specified in terms of a local relation. They have been widely studied as examples of complex dynamical systems (Gardner 1971, Wolfram 1984, Mitchell 1998), originally as examples of components in a self-reproducing machine (Burks 1970, Gardner 1970) and then within the area of artificial life (Langton 1986). In a cellular automata model, a uniform grid represents space, with each cell containing a few bits of data. Time advances in discrete steps and the laws of behavior are expressed in, say, a small look-up table, through which at each step each cell computes its new state from that of its close neighbors. Thus, the system's laws are local and uniform. In other words, cellular automata are dynamical systems where state, space and time are discrete. Cellular automata are often described as counterpart to ordinary or partial differential equations that are able to describe continuous dynamical systems. Thus, in cellular automata models, a complex system is described not using complex equations, but by simulating this system by interaction of cells following simple rules of interaction. In other words, the behavior of a complex system emerges from simple interactions of simple individuals following simple rules. Cellular automata have been extensively used in so-called artificial life-based programming where one attempts to imitate life to achieve computer-programming objectives. The benefits of Cellular Automata have made it attractive for many kinds of simulation. (Kirchener et al. 2003) have used it to simulate pedestrian dynamics. They have used the concept of friction in order to validate models of emergency egress in aircraft. (Wiemar) presents such a model to simulate enzymatic reaction networks. In his approach, he uses each lattice site as a container for one enzyme molecule, but multiple
metabolite molecules. Bezzi (Bezzi 2000) has discussed a variety of models for evolution and the immune system using cellular automata. He introduces a cellular automata model in order to study an evolving set of individuals as well as the effects of co-evolution. In particular, he discusses simulations of humoral and cellular immune responses using Cellular Automata.

***Details of other research approaches****

3 IMPLEMENTATION OVERVIEW

We embarked upon the problem of modeling and simulation the immune system from a systems biology perspective in the summer of 2002. From the beginning of the project, there has been substantial and continued collaboration of faculty and students from Computer Science, Biology and Mathematics. We have taken a software product development approach from the beginning and our current version is SIMISYS 0.3.

Our model is initial and it makes simplifying assumptions about the innate and adaptive immune systems, and the communications between the two, and simulates it at a high level of abstraction. This lays the foundation that will enable us to focus on specific issues in the immune system and model them in much richer details. We follow a systems biology approach (Kitano 2001) that requires us to model the immune system by understanding:

1. the structure of the system in terms of components and interfaces among the components,
2. the dynamics of the system and
3. the control structure.

Fig. 1 gives the high level view of the innate immune system according to our model.
Figure 1: High-level System Biology View of the Immune System

All the immune cells such as macrophages, dendritic cells, neutrophils and natural killer cells are being created in the bone marrow at certain rates. Neutrophils, the most abundant of all phagocytes, are recruited to the region of antigen attack in an infected tissue based on the concentration of chemo-attractants in the tissue. Thus, during simulation, the number of neutrophils attracted to the region is a function of the cytokine concentration produced by the macrophages in the region, which in turn depends on the number of pathogens present in the tissue. NK cells also need certain cytokines to be activated and kill the target cells.

They have been implemented as being attracted to the region of infection as a function of the lipopolysaccharide chemical (LPS) produced by bacteria. So the blood vessel is the main port of entry of the cells into the tissue where they can fight against the invader. Not only the innate players, the adaptive players also look for a suitable place to exit the blood vessels so that they can enter the lymph node where the invaders now reside once they escape from the innate fighters. The lymph vessel maintains a flow from any location in the tissue to the lymph vessel so that the immune cells and the invaders may be moved to a constricted area called lymphnode. To maintain a continuous movement of immune cells in the grid structure the blood vessels do the work of translocation of the immune cells from the lymph node to the tissue. Coordination between the blood vessels and the lymph vessel set up in the simulation has a major role in maintaining this. The complement proteins and the antibodies have been implemented as the fields of the grid position, true if they exist in that position, false in the other case. The details of their creation in the liver and the pathways by which they are activated have not been implemented yet. The current version, though, incorporates the opsonization of the bacteria and the killing of the bacteria by the MAC (membrane attack complex). The antibodies, secreted by the primed B cells, are being created in the lymph node and translocated to the tissue. There they opsonise the bacteria. The complement proteins puncture these tagged bacteria in the tissue. The current version deals only with a fixed number of complement proteins in the tissue and blood. The role of liver role in producing and continuously replenishing the complement proteins will be implemented in the next version. This completes the implementation of the basic immune cells in the current version.

The chemicals like cytokines in the blood and tissues are modeled using diffusion equations. They are loaded onto the grid cells at the location of the respective cell that secretes it. They are diffused through the whole grid depending upon their respective breakdown rate and their diffusion constants. This sets up a gradient of chemicals in the tissue. This allows for the movement of the immune cells in the tissue based on chemo attractants. This also allows the activation of certain immune cells based on the cytokine stimulation. The behavior of certain immune cells, especially macrophages depend on the state of their activation. Hence the number of immune cells is being managed by the combined action of the blood and lymph system and chemical and so is the movement of the cells.
Currently our simulation presents quite a detailed implementation of the blood vessels, lymph vessel, lymph node and tissue in which the immune cells move and interact with each other. However, the implementation of liver and bone marrow is quite simplistic at this stage.

### 3.2 SOFTWARE ARCHITECTURE

The interaction between the immune cells and pathogens has been modeled using *cellular automata*. A cellular automaton is a dynamic system whose evolution is completely described by the local interactions and is discrete in both space and time (Dewdney 1989, Toffoil 1987, von Neumann 1966, Wolfram 1986).

SIMISYS 0.3 is a complex software system with many interacting components. It has been developed keeping in mind that the system will evolve over time, as additional complexities are added to the system. A few functionalities, which have already been implemented in detail to one step further, are:

1. The MHC (Major Histocompatibility Complex)
2. The signature matching of T cells and APC
3. The signature matching of activating the T cells and B cells.

Figure 2 provides an overview of the software components that constitute SIMISYS.

The Software components are explained in detail in the next section.

![Software Architecture Diagram of SIMISYS](image)
There are three main components:

1. The Modeler
2. The Matrix
3. The Visualization Engine

The main driver program brings up a Graphical User Interface (GUI) through which the user inputs parameters to model the pathogens and the immune cells.

The emphasis of the design has been to create a highly configurable system so that specific scenarios can be modeled with ease. The parameters that can be input at this point are given in Table 1.

Once the parameters have been specified, control of the system passes to the modeler when the user presses the GO button.

The Modeler then reads the values entered by the user and it starts the simulator. The simulator creates the Matrix, all the immune cells, pathogens, blood vessels, lymph vessels and controls all the interactions among them. The Matrix models the physical space that the cells occupy. It represents the tissue and the lymphnode. It consists of a 3D grid of cells, where the simulator places the objects (all the cells and pathogens). Each entity occupies one cell. The matrix also has the ability to hold chemo-attractants and diffuse them. Currently, there are several physical spaces being modeled in the matrix: a generic tissue space, a generic lymph node, the liver, generic bone marrow, blood vessels and lymph vessels.
The visualization engine is responsible for display of the model and its simulation. The data reader reads the information from the matrix and provides the information to be displayed to the display engine. The display engine provides exchanges this information with the another graphics interface (SDL, in this case) and presents a view of the tissue, lymph or other matrices depending on the users’ interest. Either the tissue or the lymph node can be displayed at a time. A toggle key “T” on the keyboard has been provide for the user to make a choice. A separate panel for the display of the statistical results of the system has also been provided (this can be turned on or off).

4.0 SOFTWARE DETAILS

The SIMISYS Immune System simulation is implemented in C and C++. It has a multithreaded architecture based on pthreads (Nichols 1996). The graphics are displayed using SDL (Pazera 2002), a graphics library.

4.1 The Modeler Entities

We model the participants using the class structure given in Figure. A C++ object represents each entity in the simulation. A simplified class structure is given in Figure 3.

![Figure 3: Main C++ Classes Implemented](image-url)
4.1.1 Class BasicCell

Class BasicCell is at the top level of the hierarchy tree as all the cells inherit characteristics coded in this class. Common features of all the classes like `setType()`, `setStatus()`, `setState()` defined in this class. Corresponding methods of `getType()`, `getStatus()`, `getState()` are also a feature of this class. Methods like `move()`, `setGridWrapper(GridWrapper *newgw)` and `setLifeSpan(int lifespan)` are also coded here. This confines with our object oriented approach being used in the software development. A few classes at the lower level of hierarchy modify the method `move()` depending on its own specific manner in which it moves.

4.1.2 Class ImmuneCell

This class at the second level of hierarchy is the parent class of all the immune cells in the simulation. The methods defined in this class are common to all the immune cells. The main methods of interest are `hasBumped()`, `SelfNonself()` and `die()`. All the immune cells call the method `hasBumped()` when they move to another location. This method helps them check whether they have bumped into another cell and they call `SelfNonself()` to check whether it is an invader. The method `die()` is called when a cell attains its mature age. This class inherits from class BasicCell which means that immune cells which inherit from ImmuneCell class also have access to the methods of BasicCell class.

4.1.3 Class Bacteria

Currently we have a simple software model of bacteria. The class Bacteria exhibits the behavior of a bacteria once it enters the body. It starts reproducing at a specified age and moves through the tissue, travels with the flow maintained in the grid and finally reaches the lymph node by using the method `moveBacteriaLymph()`. It carries its bitsignature (expressed in its epitope) with it, which allows T cells with complimentary signature only to be activated. This further activates specific B cells and ultimately, antibodies specific for this strain of bacteria are produced. Hence the specificity of the adaptive cells is achieved. They also secrete LPS, a chemical which affects the tissue and its cells. A desirable goal is to simulate viruses, with the associated take-over of body cells. This means that genetic information should be represented in detail in all cells, and this genetic information should guide the processes taking place in the cells. A DFA for this class is

4.1.4 Class Macrophage and Class Dendritic
The class Macrophage models macrophages as phagocytes that are prevalent in the tissue in the beginning of the simulation. A fixed number of these move around at random in the tissue and exhibit their typical garbage collector behavior of eating any dead or foreign entity. The following methods describe their behavior:

A macrophage moving around in the tissue checks if it has found an entity in its vicinity by the method hasBumped(). It further checks the entity to be a foreign by the method selfNonself() implemented in ImmuneCell class. On an encounter with a foreign entity, it uses the eat() method to destroy this entity and become an APC (antigen presenting cell). The following DFA depicts its behavior:
Before presenting the antigen to T cells it processes the ingested antigen in the `chopAntigen( )` method implemented in the class cytoplasm. Just encountering an antigen does not make a macrophage an APC as normally it acts only as garbage collector. An additional cytokine signal, like IFN\(_\gamma\) secreted mainly by NK cells is needed to activate them for this. This has been incorporated by loading this chemical onto the grid by the method `loadChem( )` in the class Grid. The hyperactivated state of macrophage has also been taken care of by loading the LPS secreted by bacteria and checking its concentration in the vicinity of an activated macrophage by the method `getConcentration( )` of Grid class. Finally, the method `move( )` lets macrophage move around in the grid unless it bumps into another entity and undergoes the same processing again.

Another category of phagocytes is the Dendritic cells implemented as class Dendritic. Dendritic cells have been implemented to move to the site of infection and eat the bacteria. Normally the macrophages present in the grid perform this function but in a more rigorous situation, they are called to the site of infection. When the activated macrophages secrete TNF, dendritics know that the battle is on and they immediately rush back to the lymph node. On the way they up regulate the MHC expression and also the quantity of B7 co-stimulators. The mechanism by which this activated cell expresses the ingested antigen is as follows:

In the current version, each immune cell has a membrane marked MHC while the bacteria are marked as BHC, indicating that they are the invaders. Once the MHC of every immune cell have a groove marked on it where the broken peptides of the antigen are loaded. Whenever an immune cell is created, an instance of cytoplasm is also created in it, which carries with it a pointer of the type MHC with it. This mhc has a groove marked with an arbitrary number expressed in two bytes. When a bacteria is ingested, the antigen is chopped and chopped peptides (taken 10 + rand())%5) is expressed in two bytes. Using the method `strstr( )` of string.h, a comparison of the groove marking and the chopped antigen is made and the groove is loaded on success.

### 4.1.5 Class Neutrophil

Neutrophils, one of the first ones to reach the site of infection, have been implemented as a very simple class whereby they ooze out of the blood vessel on sensing the concentration of the IL1 secreted by activated macrophages. This means that the software simulates a scenario where the neutrophils are not called unless the battle is real intense and unless the macrophages and dendritic cells alone are not able to remove all the bacteria. It is only when the macrophages are activated by the IFN\(\gamma\) secreted by a large number of Natural Killers and that they in turn secrete IL1 to signal the resting state neutrophils flowing in the blood vessels. This is one very important detail because in normal circumstances a small number of any external body is easily cleared off by the phagocytes. Unless there is an infection of some kind, neutrophils and natural killers are not activated. They have been implemented as immune fighters with a very low life span but which arrive in large quantity. The attempt is to kill the bacteria through these innate
fighters. Due to their large number and low life span, the occupied memory is being freed for the efficiency of the system.

The general methods of the BasicCell class have been inherited by the neutrophils whereby they set their age, their type and their status when they are created. They increment their age by one unit by setAge() in each simulation cycle, check the type of the cell into which they have bumped by the selfNonself() method, kill it by kill() method and ultimately die by die(). They move around in the tissue based on the concentration gradient of INF secreted by the macrophages. This takes them to the site of infection where they can really help remove any bacteria since their number is so large. Only when the bacteria moves with the flow maintained in the tissue escape to the lymph node where they multiply, there arises a need to call other immune fighters to curb their population.

4.1.6 Class NaturalKiller

Before the adaptive immune fighters are called to the battle site, there is another category of immune cells, which is really effective in killing the immune cells. This is the Natural Killers implemented as the class NaturalKillers. The Natural Killers, known as large granulocytes, also have been implemented as coming out of the bloodvessel on sensing, LPS, a chemical secreted by the bacteria. On killing the bacteria, the cytokines (IFNγ) released by them are being loaded onto the grid. This activates the macrophages and as a result more secretion of TNF and IL12. These two chemicals are also being diffused through the grid by loadChem(). This in turn accelerates NK cells to produce even more IFNγ; hence the fact that innate fighters accelerate each other’s actions has been incorporated. The methods of interest are also mainly those that it inherits from the basic cell. It uses the method hasBumped() to check the type of the cell into which it has bumped, discriminates it on the basis of selfNonself() method of Immune cell class and kills it using kill(). Also they move in the tissue in a random fashion since they are long lived and have huge capacity to kill the invaders. The method moveNKrandom() guides their movement in the tissue.

4.1.7 Class ComplementSystem

Besides these innate players, there are also some present some proteins in the tissue and blood as the simulation starts. They are being spread all over the grid in a random manner with no gradient basis. The complement proteins are known to act as a part of the innate immunity during the initial phase of an infection. There are approximately 20 different types of them known at present. Before they can make holes using MAC (membrane attack complex), they need to be activated either by the alternative pathway or by the lectin activation pathway. Whatever pathway activates them, the initial spark comes from the C3 convertase formed by the cleaving of the protein C3. This results in further cleaving of some more neighboring C3 proteins and ultimately the chain of C5, C6, C7 and C8 getting attached to it is started. The end product of this chain is the formation of MAC. This is a high level description of the scenario of how the complement proteins work. The current simulation implements their action as innate fighters to this detail.
Further details will be added in the next release by refining these methods. Further, their opsonisation feature has also been incorporated whereby they coat the pathogens, which ease their phagocytosis. Also once the bacteria tagged by the antibodies produced by the activated B cells, finds the complement proteins in its vicinity, it succumbs to the MAC created by these proteins. This completes the link from the adaptive fighters back to the innate fighters in assisting them to kill the pathogens.

One method of interest in the ComplementSystem( ) class is inspectForNeighbors( ) whereby once the bacteria finds some complement proteins in its vicinity, the C3 protein is cleaved to form C3a and C3b. The C3b attaches itself to the bacteria to further cleave more C3 and C5 so that they can proceed to form MAC. Another level of detail which has been implemented is the fact that antibody IgM is able to activate five C1 complex of proteins at a single time. These activated complexes can initiate a cascade of events that produces a C3 convertase. On the other hand antibody IgG can activate only one C3 convertase at a time. At times other than when they are not in the vicinity of any bacteria, they move in the grid by moveComplement( ). They are very unstable so their number is being maintained constant by continuously creating more of them.

4.1.8 Class helper

The T cells are responsible for activating the B cells after they themselves get activated on bumping into an APC. The complement( ) method finds the complement string of the signature of the antigen and the basic string matching features have been used to activate T cells if a right match of the epitope of bacteria (a string of characters) and the signature of the T cell is found. The activated T cells now carry the signature of the processed antigen from the membrane of the APC. This signature is needed to prime the B cells which can create antibodies specific for an antigen.

Again only bumping does not render the T cells to be activated. Additional cytokine signal, IL1 released by activated phagocytes is needed. This is again loaded by loadChem( ) method and tested for the required concentration by getConcentration( ). Chemicals loaded at one position in the grid are spread throughout the grid by the diffuseChemicals( ) method in main. A concentration gradient is being maintained in the grid depending on the flow of body fluids in the grid. The T cells follow this gradient. Other immune fighters also follow this chemical gradient of the chemicals, which can activate them to reach the site of infection, which is compatible with what happens in any living being.
Once activated, T cells follow the flow of body fluids and travel to the nearest lymph vessel. Currently, our software model supports T cell generation and activation, but assumes a generic one whose role is integral to the generation of antibodies. The other type of T cell, the CD4 T cell that helps in antibody generation, has to be adapted to play roles in both cell and antibody mediated immunity (Kimball 1994). The T cells are activated by the APC (antigen presenting cells). But it should meet its cognate antigen peptide for that. This feature has been implemented by allocating a unique signature to every T cell that is created in the simulation. This is stored in the T cell receptor (tcr, also called the signature) field of the T cell. When a T helper cell bumps into a bacteria, it uses the find( ) method to find tcr complement in the bacteria membrane. If the match is found the T cell is activate and now it can activate the matching B cell to produce the antibodies.

4.1.9 Class Bcell

B cells created in the bone marrow are being released into the grid through the blood vessel. Our software model has antigen specific B Cells, but they are generated by a
simple random bit generator. The process of binding to specific parts of the antigen presented by the APCs is simulated in the *readBact*( ). Only the B cells that find the complement of their receptors (bcr or signature) match with the signature of the bacteria membrane are primed. When a B cell meets an activated T cell, it compares its bcr with that of the T cell using the *find*( ). This renders B cells activated. The *readTcell*( ) activates them. The DFA for Bcells is:

The primed B cells follow the flow and travel to the lymph node where they perform the clonal selection, whereby a clone of primed B cells is created. This is essential for the proliferation and cloning of the B cells since each B cell can produce only one kind of antibody. To fight against an infection we need a lot of antibodies. So once the primed B cell knows the signature of the bacteria, the activated T cell provides it the required growth factor IL2, the primed B cell proliferates to form a clone of B cells, all with the same bcr. The next stage is the career decision, whether to be a memory B cell or plasma B cell. The random bit generator forms the basis of the number of plasma B cells and the memory B cells. Now the plasma B cell takes the job of producing antibodies. Currently
the model assumes bacteria of only one type, so only one kind of antibodies are being released. We also have to recreate the process of repeated mitosis and generation of plasma cells secreting the specific receptor or antibody into the tissue.

4.2.0 Class Antibody

Two kinds of antibodies are being created in the present simulation IgM and IgG. IgM is being created in the initial phase of the simulation and IgG toward the end based on their structural advantage. In the beginning of the simulation there are more bacteria present so IgM is needed to activate five complement convertase at a time so that more bacteria may be annihilated. Towards the end IgG can very well do the job by activating just one convertase. They are being produced in large number in the lymph node by the method produceAntibodies() and are being translocated to the tissue by the blood vessels by the method searchForAntibody(). There they look for bacteria by using the method inspectForNeighbor().

4.2.1 Class Cytoplasm

Every cell, whether basic or immune, has a cytoplasm in it where all the metabolic activities vital for the survival of the cell take place. The class cytoplasm has been introduced mainly to process the ingested antigen and presenting it to the T cells. An instance of this class in each basic cell constructs the cytoplasm of each cell. Also every antigen presenting cell (APC) has MHC class II molecule (major histocompatibility molecule) in its cytoplasm. This molecule has a groove on its surface where it holds the chopped peptides of the ingested antigen. A variable mhc in the cytoplasm holds the groove in our simulated cells. The groove has markings in the form of a string of 0’s and 1’s expressed in one byte. When the macrophage or the dendritic cell ingests the bacteria, it calls the method loadMHC() whereby the chopAntigen() method creates strings of peptides, again 0’s and 1’s expressed in one byte. The numbers chosen for groove is 10 and for peptide is 10 + rand()%5 and a case of match between the two is computed. On success, mhc is loaded. The algorithm used here is:

Set the groove of mhc
Express it as a string
Do
  Chop the antigen by creating rand()%5 and add 10 to it
  Express the peptide as a string
While
  Groove is not a substring of peptide

It is to note here that although we are trying to load a peptide onto the groove but we are looking for a peptide of which the groove is a substring. The reason being that for MHCII molecules the peptide part may not totally fit into the groove. It may be even hanging out of it.
4.2.2 Class LymphVessel, Class LymphNode, Class BloodVessel

The main driving force behind the movement of the cells in the grid is backed up by the idea of two different worlds: one is the tissue and the other is the lymph node. The immune cells and bacteria move around in the grid whether in the tissue or in the lymph at random. But when the bacteria number is massive, the bacteria as well as the APC’s and immune fighters, specifically the adaptive fighters move to the lymph node where they can easily get hold of the bacteria and create antibodies to kill them with specificity. (explained in the description of class helper and class Bcell). To achieve this, the movement of the cells from one grid world to another (tissue to node and vice versa) has been accomplished through bloodvessels and lymphvessel.

There are two active bloodvessels in the grid which send new entities to the grid after they sense the chemicals around them by the inspectForChemicals(). They also suck in the activated immune fighters and the bacteria and send them to lymphvessel after they sense their presence round themselves by inspectForNeighbors(). They place these entities in the grid inside the lymph node created at the back of the main screen of the simulation (placeEntity()). The lymphvessel, located in the lymph node, looks for entities like antibodies and translocates them back to the tissue so that they can pinpoint the intruders and make the job of phagocytes and complement proteins easy.

Both these functionalities have been achieved by the use of a Gridwrapper class. It assigns a gridwrapper pointer to each entity, which lets each entity know of the world to which it belongs. Hence a continuous flow of entities is being maintained.

4.2 The Matrix

Class Grid

The basis of the simulation rests with the Grid structure. The Grid is the constituent of an array that is defined as a 3D rectangle and which makes up the world. Each grid place can contain a pointer to an entity and also store information about the current conditions in that cell. Currently we have implemented a grid with 100 grid rows, 100 grid columns and grid depth as 20. This forms the main section of the tissue where all the immune cells and move around and interact with each other. Also each grid cell maintains a concentration list that holds all the chemicals being loaded to it. Each chemical maintains the concentration of each type of chemical in it. Currently we are taking into account six chemicals, which are active in stimulating the cells and maintaining the concentration gradient, required for the movement of the cells. These are:

1. LPS (lipopolysaccharide)
2. IFN (Interferon)
3. IL-1 (Interlukin-1)
4. IL-2 (Interlukin-2)
5. TNF (Tumor Necrosis Factor)
6. IFNγ (Interferon-gamma)

Three bloodvessels and one lymphvessel are stationed in the grid as well. The lymphnode present inside the lymphvessel encloses another small grid in itself which the APC’s, T cells, B cells and the antibodies use to traverse. The smaller grid has a size of 30x30x20.

Each grid place in each world, the tissue and the lymphnode can contain a pointer to every entity and also store information about the current conditions in that cell.

![Diagram of Entity and Grid](image)

Every entity uses the idea of a grid and back end pointer. A grid pointer is used in each entity to point to the grid position that it is in the world. Using the grid pointer, a cell can check its neighboring grid positions for other entities, or inquire about the chemicals, complement proteins and antibodies present within its own position.

The back end pointer, present in each grid cell points to the cell that is in it currently. The back end pointer reveals the identity of the immune cell or the bacteria present in the neighboring positions. This allows for the cellular automaton to be useful since the immune cells can now exhibit their behaviour based on the local interactions. Further, it also helps the visualization module to get the identity of cells at the positions that it is displaying.

The main methods of this class are discussed below:

*isOccupied()* lets an entity to know whether the neighboring position is occupied or not. When we say the neighboring position, we mean anyone of the neighboring 27 positions since it is a 3-dimensional grid. The methods *setOccupied()* updates the status of each of the grid cells as it is occupied or emptied out. The method *loadChem()* allows for the loading of a specific concentration of a chemical and the method *getConcentration()* allows the access to the concentration of an already loaded chemical onto a grid location. Both these methods control the dynamics of the whole simulation.

**Class GridWrapper**
The class GridWrapper, as the name suggests, encapsulates a 3D rectangle of grid cells. It allows us to contain information about a grid along with the memory allocated to it. The GridWrapper class is essential in cases where more than one grid/world is to be simulated. Consider the case where we have to simulate a section of tissue and a lymph node. Each of this is implemented as a separate grid. We have to inform each entity that we create about the world that it belongs to, i.e., either to the tissue or the lymph node. Hence each entity structure contains a pointer to a GridWrapper class through it can access the appropriate data in the world to which it belongs.

4.3 The Visualization Engine

The visualization aspect of SIMISYS is a very important aspect of development. The focus of the project has been to differentiate itself from other modeling and simulation efforts by having a visualization aspect that can be used for educational purposes such as describing to students, the working of the immune system. Hence, we have developed a tightly integrated visualization engine that is easy to set up and can be adapted to handle introduction of entities in further releases. We also have a graphing package included in the software that can be used to decipher the results of the simulation. The engine is based on the use of SDL library (Pazera 2002), that is an open source package suitable for direct screen manipulation. The advantages of using the SDL package are the ability to blit, or paste an image on the screen at a specified location. The SDL library can be used for normal 3D operations, but it was felt that the direct blitting of images would be more to our advantage for rapid prototyping.

The engine operates as this:

1. The user specifies the images that are to be used for each type of entity present in the simulation in the initial portion of the program. The engine ignores display of any entity that has not been specified.
2. The engine then formats the image loaded for transparent background color and performs scaling of each of the entities loaded to have a series of increasingly larger images (loadImage()).

```cpp
void Engine::loadImage(string imgName, SDL_Surface *img)
{
    vector<SDL_Surface *> imgVector;

    float scaleDifference = (100.0 - this->minSize)/this->granularity/100;
    SDL_Surface *temp = sge_transform_surface(img, SDL_MapRGB(img->format, bkgdR, bkgdG, bkgdB), 0.0, 1.0, 1.0, (Uint8)0);
    SDL_SetColorKey(temp, SDL_SRCCOLORKEY, SDL_MapRGB(img->format, bkgdR, bkgdG, bkgdB));
    SDL_Surface *temp2 = SDL_DisplayFormat(temp);
    imgVector.push_back(temp2);
    float scale = 1.0 - scaleDifference;
```
for(int i=1; i < this->granularity; i++,scale-=scaleDifference)
{
    SDL_Surface *temp = sge_transform_surface(img,SDL_MapRGB(img->format, bkgdR, bkgdG, bkgdB),0.0,scale, scale,(Uint8)0);
    SDL_SetColorKey(temp, SDL_SRCCOLORKEY,SDL_MapRGB(img->format, bkgdR, bkgdG, bkgdB));
    SDL_Surface *temp2 = SDL_DisplayFormat(temp);
    imgVector.push_back(temp2);
    imgData.insert(map<string, vector<SDL_Surface *> >::value_type(imgName, imgVector));
}

3. Based on the user's key presses, the engine decides the area of the simulation that is to be displayed. There are, however, two other functions one needs to use to obtain all the information required. One must enable unicode translation by calling SDL_EnableUNICODE and one must convert SDL Key Values into something printable, using SDL_GetKeyName. Reading the keyboard events is achieved by using SDL_PollEvent. Currently six possible directions, two on each of the three dimensional axis have been implemented. This allows the user to zoom in and out of the screen as well as to move up and down the screen.

SDL_Event event;
    while(SDL_PollEvent(&event))
    {
        if(event.type == SDL_QUIT)
            done = 1;
        if(event.type == SDL_KEYDOWN)
        {
            if(event.key.keysym.sym == SDLK_ESCAPE)
                done = 1;
        }
    }
    Uint8 *keys = SDL_GetKeyState(NULL);

    if(keys[SDLK_UP])
    {
        currGW->yFocus=(currGW->yFocus>=currGW->gridCols-(SCREEN_COLS/2))? currGW->yFocus:currGW->yFocus+1;
    }
    if(keys[SDLK_DOWN])
    {
currGW->yFocus = (currGW->yFocus <= (SCREEN_COLS/2))? currGW->yFocus:currGW->yFocus-1;
    
    if(keys[SDLK_RIGHT])
    {
        currGW->xFocus=(currGW->xFocus>=currGW->gridRows-SCREEN_ROWS/2)? currGW->xFocus:currGW->xFocus+1;
    }
    if(keys[SDLK_LEFT])
    {
        currGW->xFocus = (currGW->xFocus <= SCREEN_ROWS/2)?
                        currGW->xFocus:currGW->xFocus-1;
    }
    if(keys[SDLK_PAGEUP])
    {
        currGW->zFocus = (currGW->zFocus < (currGW->gridDepth-1))? currGW->zFocus+1:currGW->zFocus;
    }
    if(keys[SDLK_PAGEDOWN])
    {
        currGW->zFocus = (currGW->zFocus > 0)?currGW->zFocus-1:currGW->zFocus;
    }

4. The engine then scans the section of the grid that has been specified. In the current version of the simulation, the engine scans a 3D section of the grid based on hard-coded values, and recognizes each of the entities present in that grid position.

void Engine::drawImg(SDL_Surface *img, int dstx, int dsty)
{
    SDL_Rect dest;
    dest.x = dstx;
    dest.y = dsty;

    SDL_BlitSurface(img, NULL, screen, &dest);
}

5. For each of the entities recognized in the grid, the engine computes the distance of the entity from the front of the screen, i.e., the closest grid position to the user by setGranularity(). Based on this distance, the image is blitted on screen such that a smaller image is pasted for an entity further than an entity closer to the user. This gives the impression of a 3D engine, without the computational expense.

Also a toggle key T enables the user to scan the lymph region.
Some of the images that have been used in the simulation are:

macrophage  neutrophil  bacteria  basic cell  NK cell  bacteria opsonized

4.4 Simulation

The algorithm used for the simulation is given below.

Read the input file and get counts for each entity
Create linked lists of each entity

For each timestamp do
    Select one of the linked lists randomly
    For each entity in the linked list do
        Run the live method of the entity
        If the entity is dead, remove it from the linked list
    End do
    If there are chemicals to be spread in the grid, spread them to neighboring cells
    Gather information about the currently viewed portion of the grid
    Pass the currently viewed portion to the display module
    Provide visual display
End do

To be more specific the program executes in the following manner:

The main driver of the software reads the input file and creates a list of each entity. The multithreaded architecture lets one of the linked lists to be selected at random and for each entity of the system, the live( ) from its corresponding class is executed. The live( ) method of each class decides the final status of the entity at the end of each simulation cycle. Of other things, the entities, which are rendered dead at the end of each simulation cycle, are removed from the list and new entities are added if they are created. Since the dynamics of the system is being controlled partly by chemicals and partly by the changes in the life cycle of the entities, the chemicals released by the entities are diffused through the grid on a regular basis. One after the other, depending upon which thread
gets the control, each linked list of entities goes through the simulation cycle. The display engine also gets the control on a regular basis. It gathers the information about the currently processed portion of the grid, updates the data and passes the information to the display module which provides the visual display.