Discrete nondeterministic modeling of biochemical networks

John R. Jack
Louisiana Tech University

Follow this and additional works at: https://digitalcommons.latech.edu/dissertations
Part of the Bioinformatics Commons, and the Computer Sciences Commons

Recommended Citation
https://digitalcommons.latech.edu/dissertations/429

This Dissertation is brought to you for free and open access by the Graduate School at Louisiana Tech Digital Commons. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of Louisiana Tech Digital Commons. For more information, please contact digitalcommons@latech.edu.
DISCRETE NONDETERMINISTIC
MODELING OF BIOCHEMICAL
NETWORKS

by

John R. Jack, B.A., M.A.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

May 2009
We hereby recommend that the dissertation prepared under our supervision by John Jack entitled
Discrete Nondeterministic Modeling of Biochemical Networks be accepted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computational Analysis and Modeling.

Supervisor of Dissertation Research
Head of Department
Advisory Committee

Approved:
Director of Graduate Studies
Dean of the Graduate School
Dean of the College

GS Form 13a
(6/07)
ABSTRACT

The ideas expressed in this work pertain to biochemical modeling. We explore our technique, the Nondeterministic Waiting Time algorithm, for modeling molecular signaling cascades. The algorithm is presented with pseudocode along with an explanation of its implementation. The entire source code can be found in the Appendices. This algorithm builds on earlier work from the lab of Dr. Andrei Păun, the advisor for this dissertation. We discuss several important extensions including: (i) a heap with special maintenance functions for sorting reaction waiting times, (ii) a nondeterministic component for handling reaction competition, and (iii) a memory enhancement allowing slower reactions to compete with faster reactions.

Several example systems are provided for comparisons between modeling with systems of ordinary differential equations, the Gillespie Algorithm, and our Nondeterministic Waiting Time algorithm. Our algorithm has a unique ability to exhibit behavior similar to the solutions to systems of ordinary differential equations for certain models and parameter choices, but it also has the nondeterministic component which yields results similar stochastic methods (e.g., the Gillespie Algorithm).

Next, we turn our attention to the Fas-mediated apoptotic signaling cascade. Fas signaling has important implications in the research of cancer, autoimmune and neurodegenerative disorders. We provide an exhaustive account of results from the Nondeterministic Waiting Time algorithm in comparison to solutions to the system
of ordinary differential equations described by another modeling group. Our work with the Fas pathway led us to explore a new model, focusing on the effects of HIV-1 proteins on the Fas signaling cascade. There is extensive information in the literature on the effects of the HIV-1 proteins on this pathway. The model described in this work represents the first attempt ever made in modeling Fas-induced apoptosis in latently infected T cells.

There are several extensions for the Fas model discussed at the end of the work. Calcium signaling would be an interesting avenue to investigate, building on some recent results reported in the literature. For the HIV model, there are several extensions discussed. We also suggest a new direction for the Nondeterministic Waiting Time algorithm exploring parallelization options.

Keywords: Discrete, Stochastic, Nondeterminism, Fas, Apoptosis, HIV, Computational Biology, Systems Biology.
APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author

Date 5-13-09
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical examples of biochemical reactions</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>An example system to illustrate memory enhancement</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Reactions involving translocation of the HIV genome and integration into the host’s genome</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>Reactions involving HIV long terminal repeat (LTR) and HIV mRNA production</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Translation and degradation rules for HIV mRNA</td>
<td>89</td>
</tr>
<tr>
<td>5.4</td>
<td>Reactions involving Tat protein</td>
<td>90</td>
</tr>
<tr>
<td>5.5</td>
<td>Reactions involving Vpr protein</td>
<td>91</td>
</tr>
<tr>
<td>5.6</td>
<td>Reactions involving HIV protease</td>
<td>91</td>
</tr>
<tr>
<td>5.7</td>
<td>Reactions involving Nef protein</td>
<td>92</td>
</tr>
<tr>
<td>D.1</td>
<td>Kinetics rates for the Fas model</td>
<td>183</td>
</tr>
<tr>
<td>D.2</td>
<td>Additional kinetic rates for HIV-1-infected model</td>
<td>186</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Example of a Membrane System ........................................ 17
Figure 2.2 Comparison of the graphical representation and SBML code for a model generated using the CellDesigner software .................... 28
Figure 2.3 The two structs: Protein and Reaction nodes used to build the Alphabet and Rule sets of the Membrane System ...................... 30
Figure 2.4 Heap example ................................................................. 37
Figure 2.5 Recalculation of the Waiting Time for reaction E .................. 39
Figure 2.6 Reheaping the applied reaction ........................................ 40
Figure 2.7 Recalculating the Waiting Time for reaction I ..................... 41
Figure 2.8 Reheaping reaction I ....................................................... 42
Figure 2.9 Reheaping reaction J ....................................................... 43
Figure 2.10 The number of $C$ molecules for a simulation of $t = 10$. The NWT algorithm results are represented with bars while the solution to the system of ordinary differential equations is the solid line .......... 49
Figure 2.11 Results of the memory enhancement simulation for the ordinary differential equations and the NWT algorithm. (a) molecules of $B$ and (b) molecules of $C$ ................................................. 51
Figure 3.1 The Lotka-Volterra model ............................................... 57
Figure 3.2 Results of three simulation techniques for the Lotka-Volterra model (up to 100 seconds). (a) solution to ordinary differential equations, (b) the NWT algorithm, and (c) the Gillespie Algorithm ............... 58
Figure 3.3 Results of the two simulation techniques for the Lotka-Volterra model (up to 500 seconds). (a) solution to ordinary differential equations and (b) the NWT algorithm ........................................ 59
Figure 3.4 Picture of the circadian rhythm model ................................ 61
Figure 3.5 The results for the circadian rhythm model: (a) the NWT algorithm and (b) the Gillespie Algorithm. Both algorithms are plotted against the solution to the system of ordinary differential equations.

Figure 4.1 Picture of the Fas-mediated apoptotic signaling cascade. Both the type I and type II pathways are illustrated.

Figure 4.2 The decline of full length Caspase-3 for different concentrations of Bcl-2: (i) baseline, (ii) 10-fold increase, and (iii) 100-fold increase. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.3 The decline of full length Caspase-3 for decreased concentrations of Bcl-2: (i) 10-fold and (iii) 100-fold. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.4 The effects of Bcl-2 binding to Bax only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.5 The effects of Bcl-2 binding to tBid only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.6 The effects of Bcl-2 is binding to Bid only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.7 The results of baseline Bcl-2 concentration with each of the four mechanisms for Bcl-2 inhibition (binding with Bax only, Bid only, tBid only, or Bax and tBid). (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.8 Investigation of the effects of Bcl-2 increase (100-fold) for the type I pathway. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.9 Results of the three simulation techniques and experimental data provided by [47], showing decline of (a) full length Caspase-3 and (b) full length Caspase-8.

Figure 5.1 Estimated number of HIV-1-infected individuals in 2007 as well as the various HIV-1 subtypes per region.
Figure 5.2 HIV-1 protein effects on the Fas-mediated signaling cascade ...... 86

Figure 5.3 (a) Tat protein upregulates Caspase-8 levels by three-fold. (b) Vpr upregulates Bcl-2 and downregulates Bax by 30% and 20%, respectively ................................................. 93

Figure 5.4 (a) Total reduction of full length Caspase-3 is seen after ~ 40 hours in the latent model, whereas the nonlatent model takes ~ 47 hours. (b) The decline of Bid through interactions with Caspase-8, leading to a rise in tBid .................................................. 94

Figure 5.5 Interactions involved in the type I and type II pathways for the non-latent simulation. (a) The results for the three days of simulation and (b) An excerpt of one minute from the three day simulation (from 32 hours to 32 hours and 1 minute) ........................................... 94

Figure 5.6 Interactions involved in the type I and type II pathways for the non-latent simulation. (a) The results for the three days of simulation and (b) An excerpt of one minute from the three day simulation (from 32 hours to 32 hours and 1 minute) ........................................... 95

Figure 6.1 The effects of an HIV-1-infected T cell on its neighboring healthy, bystander cells ..................................................... 99

Figure 6.2 The picture represents how calcium relates to the proteins of the Fas-mediated apoptotic signaling cascade ............................................. 102

Figure 6.3 A proposed reaction network for the partitioned system ................ 105

Figure 6.4 Partition the cell into subcompartments for parallel simulation .......... 106

Figure 6.5 A different view at breaking the system into subcompartments ........... 107
ACKNOWLEDGEMENTS

There are numerous people whom I wish to thank for making the writing of this dissertation possible. My parents have provided endless inspiration and support for me throughout the years. My sister has sat through endless rants as I have climbed the academic ladder. Also, I would like to thank my brother for helping me understand when to speak and when to listen.

Many friends have supported me along my way. Joe Callan, my oldest friend, has provided deep philosophical grounding to me, and is always up for a healthy debate. Katherine Fusco helped me get back on track when I needed it most. Joe Calabrese has listened to my frustrations and helped me focus. Jim and Kaitlin McNamara are two of my dearest friends in Ruston. Dorene Kordall is always willing to lend a helping hand. Krystal Skye helped me weather the storm of the dissertation writing.

I would also like to thank my dissertation committee – Dr. Galen Turner, Dr. Vir Phoha, Dr. David Mills, and Dr. Mark DeCoster. I have been lucky enough to take at least one class with each one of these men; their talents and support inside and outside the classroom have made this work possible.

Finally, I would like to thank Dr. Andrei Păun. Without him, this work would not exist. He gave me endless intellectual challenge; opportunities for funding; a way to view research from a “big ideas” perspective; and, most important, friendship.

There are countless others who have helped. I would like to thank them as well.
CHAPTER 1

OVERVIEW OF THE LITERATURE

The focus of this dissertation is the simulation of molecular signaling cascades, specifically the Fas-mediated apoptotic pathway. The discussion centers around living systems – e.g., biological cells – and the changes in their biochemical compositions, which is brought about through the various interactions of the intracellular proteins. Indeed, signal transduction (and molecular signaling cascades) describes the systematic interactions of different cellular proteins, starting with some sort of signal (e.g., an external ligand binding to a cell surface receptor) and reaching some sort of endpoint (e.g., the upregulation of a protein eliciting physical changes to the cell). Our group wishes to explore the molecular mechanisms behind biochemical evolution with implied physiological responses through computational modeling.

We are at the start of a new millennium and a (relatively) new science. We are beginning to understand ourselves on a level virtually incomprehensible at the beginning of the last century. Humanity is on the brink of incredible technological breakthroughs – the power to engineer/manipulate our own genomic data, the stuff that makes up who and what we are. However, as with any engineering project, we will need computational models to be successful. Genetic-based manipulations, changing the biochemical nature of ourselves (and our cells) for the betterment of mankind, will
require an exceptional degree of certainty, which is only knowable through computer-driven modeling. We need reliable and predictable circumstances as we go diving into our own genome.

The human genome project has given us a rich text on humanity. The initial draft sequence of the human genome was first reported in [67]. So, we now have the sequence, but we lack any sort of deep understanding of it. With our current level of knowledge, the situation is analogous to handing a tome written in Japanese to a person who understands only English and requesting that they translate the text. There are still many uncertainties about DNA – for instance, the precise number of genes encoded in the human genome. At first, scientists estimated that there could be two million genes in the entire human genome. The results of [67] put the number of genes at 30,000-40,000. Now, it seems like there are more like 20,000-25,000 genes. Indeed, only 2% of the human genome is believed to encode for genes.

There has been an enormous thrust for computer science to address and understand the relevance of the genomic biological data. Besides the modeling of molecular signaling cascades discussed herein, there is a large thrust into protein-folding prediction. Computer algorithms have been designed to decipher the three-dimensional folding of a protein based on its amino acid sequence. Understanding this structure will give insight into the function, since these two elements – structure and function – are intrinsically linked. With the sequencing of the entire human genome, scientists are looking to computer models in order to understand all that the Human Genome Project and similar efforts have uncovered for us.

Although the number of genes encoded in a cell is alarmingly small, given the vast
physical differences in the life of this planet, we look for another aspect to explain the diversity of living things. To understand ourselves, we need to quantify the interactions of the things encoded by the genes. The complexity of the machinery we call life stems from relatively basic components (proteins) interacting through intricate reaction networks. Simplicity breeds complexity.

Consider the picture of a famously studied biochemical network: the EGFR network [86]. Looking at the picture of this network diagram, there is a disquieting moment when we realize how intensely complex and intricate even one cellular signaling pathway can be. We are struck with a sense of awe, when we try to imagine a network diagram illustrating all of the signaling pathways at work in a normal, healthy functioning cell. Dysfunctional cells – for instance, a cancerous cell or one infected by a retrovirus – does not paint a picture any less complicated.

The scientific community is right now attempting to unlock the molecular mechanisms underlying the functionality of the scariest and deadliest of diseases and disorders. We do not yet understand how all of the puzzle pieces fit together. We are only beginning to uncover and understand the components behind the most complicated – in terms of finding a cure – diseases and disorders. Discovering the methods to cure diseases, such as cancer, will require an unprecedented degree of cooperation between computer scientists, mathematicians, and biologists.

In the twentieth century, we saw an explosion of breakthroughs in the physical sciences. With the help of mathematicians and computer scientists, we were able to unlock incredible mysteries from the very large (black holes, stars, and the planets) to the very small (harnessing the power of the atom). We have even begun venturing to
other planets. However, we are now on the cusp of a new thrust in mathematics geared towards assisting biologists. As we turn our eyes towards the implications of genome sequencing, nanotechnology, DNA computing, and gene therapy, we see a future where predictions by computer models are a very important aspect in a fantastic new realm of science, leading to breakthroughs in therapies to fight diseases, aging, or any other ailments associated with the biochemistry of life.

For the rest of Chapter 1, we will introduce some of the existing techniques for modeling the dynamics of intracellular proteins in a reaction network. Differential equations have been the predominate form of modeling for a very long time. However, a promising algorithm was developed in the late 1970s using stochastic fluctuations to more accurately predict protein dynamics. Since then, both methods have been improved, developed, modified, adapted and even combined to give us fast, accurate simulations of molecular signaling cascades.

1.1 Modeling with Differential Equations

Systems of ordinary differential equations are employed to model a wide range of phenomena, including but not limited to modeling the dynamics of molecules in a biochemical reaction network. In fact, due to their simplicity and the speed in which they can be solved, systems of ordinary differential equations are quite popular in the modeling of molecular signaling cascades. However, a recurrent theme in this work will be the fact that molecular modeling with ordinary differential equations does not always yield desireable results. Moreover, the solutions to systems of ordinary differential equations can often yield misleading results, failing to accurately represent
the minority behavior of a few cells in favor of results illustrating the average behavior of the majority of cells. In later chapters, we will continue to comment on this shortcoming of the differential equations. For now, we will briefly discuss how we can model biochemical systems with systems of ordinary differential equations.

We set up an ordinary differential equation for each type of molecule in the system. For each species $X_i$, we have

$$\frac{dX_i}{dt} = f_i(X_1, \ldots, X_n), \quad (1.1)$$

where $f_i$'s are functions (possibly nonlinear and nonhomogeneous). For example, the Hill function, which was initially developed to describe the binding of oxygen to hemoglobin [46], is a classic nonlinear function now having widespread use in describing cooperative binding (such as ligands binding to receptors). In fact, many biological phenomena are being modeled with nonlinear functions.

Essentially, to model the dynamics of protein interactions, you will need a differential equation for each protein with the functions defined according to how the proteins react to each other in the system. For example, we consider an early investigation into chemical equilibrium, which was made by L. Wilhelmy [75, 127]. His studies were focused on the following sucrose reaction:

$$H_2O + C_{12}H_{22}O_{11} \rightarrow C_6H_{12}O_6 + C_6H_{12}O_6. \quad (1.2)$$

If we let $S(t)$ represent the concentration of sucrose, then Wilhelmy was able to show that

$$-\frac{dS}{dt} = kS, \quad (1.3)$$
where $k$ is a kinetic rate constant. He designed this equation to match the empirical evidence that the rate of decrease of sucrose concentration was proportional to the concentration remaining unconverted (the law of mass action discussed further in Chapter 2).

If we let $S_0$, represent the initial concentration of sucrose, then we have

$$S(t) = S_0 e^{-kt}. \quad (1.4)$$

Since the time of Wilhelmy, chemical kinetics have received a great deal of attention. To address interesting problems in the biochemistry of life, we need much more complex systems. These systems will involve many reactions with a large degree of interdependence. Indeed, we will see increasingly complex systems of ordinary differential equations in the coming chapters of this work. This complexity will be driven by the number of interacting elements and reactions, but not necessarily from complicated nonlinearity such as the Hill Function. With more complex systems, we will need to approximate the solutions to the systems of ordinary differential equations. Once the system of differential equations is written out, mathematically describing the molecular mechanisms, it is time to apply approximation methods.

The most common method for approximating a system of ordinary differential equations is the fourth order Runge-Kutta method. This is the method employed by a large majority of computational biology labs. Often, MATLAB is used, and the `ode23` and `ode45` solvers are built-in functions utilizing Runge-Kutta to provide an approximate solution to the system of ordinary differential equations. For an example of the MATLAB code we have used to determine solutions to systems of ordinary
differential equations, we refer the interested reader to Appendix A (describes the circadian rhythm model from Chapter 3).

We have hinted that differential equations are not the only way to model biochemistry. We will now briefly discuss the stochastic techniques for modeling biochemical reaction networks.

1.2 Stochastic Methods and the Gillespie Algorithm

There are situations where ordinary differential equations fail to adequately represent cellular populations. The biochemical reason for this usually stems from situations of low molecular multiplicity. In one of the most important works on stochastic approaches to chemical kinetics, McQuarrie [75] provided a rich description of the historical background to stochastic techniques as well as some exactly solvable systems.

Kramers was the first to use stochastic ideas for modeling the kinetics of chemical equations [63, 79]. The idea of stochastic approaches for modeling chemical systems revolves around the *chemical master equation*. This equation describes the probability of every possible state of the cell (with respect to biochemical composition). Instead of a differential equation for each protein, one would essentially have a differential equation for every possible state of the cell. For very small systems, the chemical master equation can be solved directly (see [75] for some systems). However, it becomes difficult or impossible to directly find the chemical master equation for systems of nontrivial size. It is this reason which led D.T. Gillespie to formulate his now ubiquitous algorithm for exactly solving the chemical master equation.
Gillespie published two landmark papers in 1976 and 1977. In [35], he presents the framework for an exact stochastic method, which accurately predicts the chemical master equation. Then, in [36], Gillespie describes the Stochastic Simulation Algorithm (SSA); the algorithm is now aptly named the Gillespie Algorithm, and it is the most commonly applied/adapted technique for stochastic simulation of biochemical networks. The Gillespie Algorithm is at the heart of most discussions on stochastic modeling.

In [36], Gillespie discusses two important points on the failure of classical modeling (differential equations); the approach assumes that the time evolution of a chemically reactive system is both continuous and deterministic. However, in nature, chemically reacting systems evolve in a discrete manner, since molecular multiplicities can obviously change only by integer amounts. Also, it is impossible to predict the future molecular population levels through deterministic systems (a system of ordinary differential equations), because we cannot know the exact positions and velocities of all the molecules in the system. Hence, time evolution for simulations must be a nondeterministic process, in order to account for all of the possibilities.

We will forgo an explanation of the algorithm, since it has been reported ad nauseam in the literature. However, we would like to mention the main limitation of the algorithm. As stated in the original paper [36], the Gillespie Algorithm places a high premium on the speed of the computer's CPU. The limitations are dependent on the number of reactions in the system. Also, the algorithm requires multiple runs to correctly quantify the system. This works in conjunction with the speed limitations, making stochastic simulations an enduring process.
1.3 Improving the Gillespie Algorithm

Since its creation in 1977, the Gillespie Algorithm has been the focus for improvements in efficiency. The most notable improvement for the Gillespie Algorithm comes from the work of Gibson and Bruck [34]. They were able to reduce the computational complexity of the algorithm considerably, through the addition of a method for sorting the reactions and reducing the dependence on random number generation. However, the limitation associated with reaction network growth – number of molecules and reactions – is still an issue.

A number of methods have now been proposed to combine differential equations with the Gillespie Algorithm. These hybrid methods attempt to divide the reactions into fast and slow. We will not provide an exhaustive discussion on each method, but we wish to mention two notable works below.

The work of Haseltine and Rawlings [43] has been well-cited. They provide the theoretical background for dividing reactions into fast and slow subsets, allowing for the fast reactions to be approximated either deterministically or as Langevin equations. Essentially, they are able to integrate the system over much larger time steps than the original Gillespie Algorithm. For the original Gillespie Algorithm, increasing the number of molecules for a fast-reacting protein will significantly increase the computational load; however, by using deterministic processes for fast reactions, the computational load of their algorithm will not increase in this case.

Rao and Arkin [101] applied the quasi-steady state assumption to modify the Gillespie Algorithm. Using the quasi-steady state assumption, they were able to reduce model complexity by reducing the number of molecular species and reactions.
Essentially, the assumption is that the net rate of formation is approximately zero for highly reactive and transitory species – e.g., enzyme-substrate complexes. In their paper, the authors provide some mathematical rigor behind the algorithm as well as some results for example systems.

1.4 Our Work

In Chapter 1, we have provided an introduction to the concept of modeling biochemical reaction networks and a brief discussion on some of the most popular techniques. For the rest of this work, we will focus on biochemical modeling with our algorithm, the Nondeterministic Waiting Time algorithm.

In Chapter 2, we will introduce the Nondeterministic Waiting Time algorithm. We provide an in-depth look at the pseudocode; a discussion on the implementation; and some results from example systems illustrating the concepts of the algorithmic improvements is provided in the chapter. Chapter 3 illustrates the results of two popular models: the Lotka-Volterra predator-prey model and a circadian rhythm model. The two models are used to emphasize the difference between our algorithm and the solutions to systems of ordinary differential equations and the Gillespie Algorithm.

In Chapter 4, we provide simulation results for the Fas-mediated apoptotic pathway. We compare the results of our algorithm for the model with the solutions to the systems of ordinary differential equations provided in [47]. Then, in Chapter 5, we propose an extension of the Fas model discussed in Chapter 4. The extension revolves around the addition of proteins encoded in the human immunodeficiency virus. The model, which explores T cell latency, is the first reported of its kind. Chapter 6
provides the conclusion for the work. A discussion on possible research extensions is also provided therein.
CHAPTER 2

THE NONDETERMINISTIC WAITING TIME ALGORITHM

In Chapter 1, we provide a survey – but in no way an exhaustive account – of the numerous algorithms pertaining to the simulation of molecular interactions in a biochemically reactive system. Each of the simulation techniques discussed have their own particular strengths and weaknesses. For the stochastic techniques, the strengths and weaknesses typically revolve around the accuracy with which the algorithm can predict the chemical master equation and the computational efficiency with which it functions. For the deterministic techniques, speed is one of the main strengths, but at a loss of the randomness inherent in living biochemical systems. Predicting the behavior of the minority populations is impossible with deterministic techniques, since they display the average behavior of the system and, thus, favor the majority behavior.

The focus of Chapter 2 will be the discussion of a different type of biochemical simulation algorithm. A technique designed in such a way that it is capable of exhibiting behavior similar to continuous deterministic approaches for certain biochemical models, but it behaves similar to the discrete stochastic approaches – e.g., the Gillespie Algorithm – for certain other types of systems. We have chosen to call
this technique the Nondeterministic Waiting Time (NWT) algorithm.

This work is an extension of previous modeling efforts initiated by the advisor for this dissertation, Dr. Andrei Păun. In [17] the groundwork for a Deterministic Waiting Time algorithm was laid out. Here, we provide an efficient, refined algorithm with several important extensions from the previous algorithm. Notably, the algorithm now has a nondeterministic component and a memory enhancement, which gives it a unique ability to simulate reaction competition over limited numbers of reagents. Additionally, the implementation of a min-heap with special maintenance functions improved the efficiency of the previous simulation algorithm.

In Chapter 2, we will introduce the concepts behind the Nondeterministic Waiting Time algorithm. The entire source code is available in the Appendices (B and C) near the end of this work. However, in Chapter 2, we will provide several blocks of pseudocode as necessary to discuss the important aspects of the simulation technique. After introducing the algorithm, we will provide a couple of small example systems to illustrate the particular concepts and enhancements which we have implemented in our NWT algorithm. We must begin, however, with a discussion on Membrane Systems and why we have chosen this computing paradigm as the framework of our biochemical reaction network simulator.

### 2.1 Introducing Membrane Systems

In Section 2.1, we will present the foundations for our discrete, nondeterministic biochemical simulation technique: the Nondeterministic Waiting Time algorithm. In the design of this algorithm, our goal is to define a simulation technique between the
realm of the Gillespie Algorithm and modeling with systems of ordinary differential equations. Moreover, we wish to have a modeling technique which is less computationally intensive than the Gillespie Algorithm, yet maintains a level of nondeterminism (or stochasticity) which set it apart from solutions to systems of ordinary differential equations. To describe the foundations of the NWT algorithm, we must explore the realm of a relatively new paradigm of computing: Membrane Systems (or P Systems).

The evolution of DNA, RNA, and proteins during life's tenure on Earth is the story of the storage and application of information, similar in development to the field of computer science. For these macromolecules of life, there is a classic debate between life scientists surrounding which of them evolved first. Implicit to this debate is an emphasis on the billions of years of information theory inherent in life and nature.

Let us assume, for the sake of argument, that RNA existed before the others. Then we can imagine that the initial molecules of life had the ability to store information, the way modern messenger RNA (mRNA) carries the genetic information from the nucleus to the cytoplasm for translation into a protein. However, we can also imagine some initial molecules of life possessing the ability to put the information to work, the same way small nuclear RNA (snRNA) has responsibilities in the modern cells pertaining to the transcription of a gene. It is from the incredible achievements in information storage and application, apparent in all living cells, that Membrane Systems evolved as a way to view the molecular activity within a cell as a computation.

The concept of computing with membranes was first proposed in 1998 by Gheorghe Păun [100]. As a model for computation, Membrane Systems have proven to be quite useful; they take advantage of exponential space in order to solve computationally
hard problems efficiently. In their short history as a computational paradigm, a multitude of Membrane Systems have been proposed. These Membrane Systems have the ability to attack NP-Complete problems through the use of exponential space, sharing some of the fundamental concepts of biological parallelism as DNA computing [1]. For example, a Membrane System has been described which is capable of solving the boolean SAT problem in linear time [133]. However, no one has yet been able to build an actual Membrane Systems computer out of a cell. If we could harness the computational power of a cell, we could break through the glass ceiling on efficient solutions to computationally complex problems – i.e., NP-complete. Although the majority of Membrane Systems research has been on abstract models and theory, there are a few groups who wish to use Membrane Systems in a different way. Some computer science groups are investigating the use of Membrane Systems to address problems in computational biology. This is the direction of our interest.

For our purposes, we will define a Membrane System, \( \Pi \), in the following way:

\[
\Pi = (\Sigma, L, \mu, M_1, ..., M_m, R_1, ..., R_m),
\]

(2.1)

where

- The alphabet, \( \Sigma \), is a list of all the proteins in the system. The alphabet does not contain information on the numbers of each protein. That is, the alphabet is initialized with only the number of different types of proteins with no information on the total number of molecules in the system.
- A set of labels, \( L \), representing all of the different compartments of the system.
- The membrane structure, \( \mu \), represents the hierarchical organization of the different compartments, \( L \).
• The multiplicity sets, $M_i$ where $1 \leq i \leq m$, contain the multiplicities of the proteins within each compartment – i.e., the number of molecules per protein.

• The rule sets, $R_i$ where $1 \leq i \leq m$, contain the rules associated within each compartment. The rules are the chemical reactions.

It is worth mentioning that for the remainder of this text, reaction/rule and species/protein will be used interchangeable. One of the nice features of Membrane Systems is that they can easily be comprehended using a graphical representation. For an example, let us consider a Membrane System illustrated in Figure 2.1.

In mathematical terms, we describe the system below

- $\Sigma = \{a, b, c, d, e\}$
- $L = \{1, 2, 3\}$
- $\mu = \{1, 2, 3\}$
- $M_1 = \{ac\}, M_2 = \emptyset, M_3 = \emptyset$
- $R_1 = \{e \rightarrow e_{\text{out}}\}, R_2 = \{b \rightarrow d, d \rightarrow de, (cc \rightarrow c) > (c \rightarrow \delta)\}, R_3 = \{a \rightarrow ab, a \rightarrow b\delta, c \rightarrow cc\}$.

The Membrane System described above has eight rules and five distinct types of proteins. N.B., the $\delta$ in the rules denotes a “dissolve” rule whereby the membrane in which the rule is executed disappears – all proteins present in the membrane at this time are dumped into the parent membrane. Notice, the system begins with only one protein of $a$ and one protein of $c$, both of which are present in membrane 3.

The rules of the system govern the biochemical evolution of the system. The rules act in a maximally parallel manner – i.e., for the system to evolve from one state to
the next state, all rules which can be applied are applied. For abstract Membrane Systems, the transition from one system state to the next is called a computation. The computations of a Membrane System are similar to the transitions of a Turing Machine. The system continues until it reaches a stopping configuration. For the example system above, the first computation would yield \( cc \) from \( c \) and \( ab \) from \( a \) (applying the only two possible rules).

For our NWT algorithm, we would like to use Membrane Systems more as a data structure. The simulation of a chemically reactive system – e.g., a living cell
is merely the evolution of the Membrane System according to the set of rules, \( \{R_1, \ldots, R_m\} \). To do so, we will need a discussion on how/when rules will execute. As the rules describe the interactions between proteins in the alphabet, the evolution of the Membrane System tracks protein dynamics. While there are many types of biochemical reactions, we will list a few of these to facilitate our understanding on the design and implementation of the Membrane System proposed in Equation 2.1. Some basic examples of biochemical reactions are listed in Table 2.1.

Table 2.1 Typical examples of biochemical reactions.

<table>
<thead>
<tr>
<th>Reaction Type</th>
<th>Reaction Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 ): Monomolecular decay</td>
<td>( A \xrightarrow{k_d} \emptyset )</td>
</tr>
<tr>
<td>( R_2 ): Monomolecular reaction</td>
<td>( A \xrightarrow{k_x} B )</td>
</tr>
<tr>
<td>( R_3 ): Bimolecular reaction</td>
<td>( A + B \xrightarrow{k_y} C )</td>
</tr>
<tr>
<td>( R_4 ): Trimolecular reaction</td>
<td>( A + B + C \xrightarrow{k_z} D )</td>
</tr>
</tbody>
</table>

In order for the Membrane System to illustrate protein dynamics over time, we need to discuss the temporal aspects of the rules of the system. To model the biochemistry of life, the individual chemical reactions described in the system occur must over different lengths of time in an asynchronous manner. The rules of our Membrane System obey the law of mass action, which was first formalized in 1864 [37, 38, 120]. The law of mass action states that a reaction rate is directly proportional to the number of reactants available in the system. In other words, the time a reaction takes to occur is dependent on the number of its reactant molecules.

With the law of mass action, we have a way to associate time dynamics with the evolution of the Membrane System as it jumps from one configuration to the next. We also have a way to make the rules occur in an asynchronous manner. As previously
mentioned, a Membrane System typically evolves by applying rules in a maximally parallel manner. In other words, when the system jumps from one configuration to the next, any and all rules which can be applied (given sufficient reactants) are applied. But, when we use the law of mass action, the reactant concentrations govern how much time must transpire before a particular reaction can take place. Therefore, for any particular configuration of the Membrane System, the number of reactant molecules for a given reaction determines when that reaction is next slotted to occur. The values associated with the law of mass action are called *kinetic rates*.

These kinetics rates must be determined through biological experimentation. As such, the kinetics of a chemically reactive system are often described as concentration-based values. The reason for this is the fact that biological results are often generated from enormous populations of cells. Often a biological experiment will consider a population of millions of cells. To determine the intracellular concentrations of proteins, these cells are then lysed as a large population. The intracellular molecules are then measured in terms of light intensity (radiological or photonic markers), which gives data on general concentrations of particular molecules across the population. Finally, the values are averaged to give the concentration per cell. Therein, lies a major problem with biochemical modeling. We rely on the values generated in the biological lab, and these values are often generated over entire cell populations instead of individual cells. Hence, the interesting phenotypic, biochemical and physiological characteristics of individual cells can be lost in lieu of the behavior of the majority of the cells in the population.

There are techniques which can measure single cell dynamics, and this technology
is very promising. For instance, important results on p53 have been reported [65] from the results of measuring single cell dynamics instead of averaging over cellular populations. The authors of [65] were able to show individual cells undergo not dampened oscillations, as previously reported [5], but each individual cell instead undergoes different numbers of oscillations. The average behavior for the cell population appears dampened, but individual cells do not.

At the lab of Dr. Mark DeCoster of Louisiana Tech University, we now have the capabilities to measure single cell dynamics via a high-speed imaging system. Towards the end of the work of this dissertation, Chapter 6, there is discussion on the dynamics of Fas involving Ca\(^{+}\) signaling, which could be explored using the system at Louisiana Tech University. It is our hope that future collaborations between Dr. Andrei Păun’s computational group and Dr. DeCoster’s biomedical laboratory will help unlock some of these secrets and provide new aspects to Fas signaling cascade. Regardless of whether data comes from large cell populations or single cell dynamics, we, as modelers, must remain vigilant and build the best models with the data available to us. As is the case with the model discussion of Chapter 5, expanding the Fas model with the effects of viral proteins, relying on biological data from a wide variety of sources can result in difficulties when trying to set up a model.

The kinetic rates, \(k_R\) for some reaction \(R\), will often have units based on nMs, \(\mu\)Ms, etc. Let’s assume we have these kinetic rates for every reaction in our Membrane System. We have already stated that our technique is a discrete one. Therefore, in order deal with multiplicities of proteins as opposed to concentrations, we must calculate a discrete kinetic constant. This discrete kinetic constant will be based on numbers of
molecules. When we initialize the Membrane System, we must calculate the discrete kinetic constants from the concentration-based kinetic rates in the following way

\[ \text{const}_R = \frac{k_R}{V^{i-1} \times N_A^{i-1}}, \]  

(2.2)

where \( V \) is the volume of the system, \( N_A \) is Avogadro's constant \((6.0221415 \times 10^{23})\) and \( i \) is the number of reactants involved in the reaction.

With the law of mass action and the discrete kinetic constants, we have the means to allow the rules of the Membrane System to occur at times dependent on reactant multiplicities, which are subject to variation throughout the entire simulation run. We can now define a reaction's Waiting Time \((WT)\). The Waiting Time is a value associated to each reaction, signifying when the next time a single instance of the reaction will occur. As molecular multiplicities will change throughout a simulation, so will the reaction Waiting Times (in accordance with the law of mass action).

As mentioned, we initialize the discrete kinetic constants at the beginning of the simulation. Once we have these constants (along with the initial multiplicities of the proteins), we can initialize the Waiting Time for every reaction in the system. The Waiting Time is the amount of time required for the execution of one instance of a reaction. For a first order equation, like \( R_1 \) from Table 2.1, the Waiting Time is calculated with the following equation:

\[ WT_{R_1} = \frac{1}{k_d \times |A|}, \]  

(2.3)

where \( A \) is the reactant required for reaction \( R_1 \), \( |A| \) represents the number of molecules present in the system at the moment of Waiting Time calculation, and
$k_d$ is the discrete kinetic constant. N.B., $R_1$ and $R_2$ from Table 2.1 are calculated the same way (if we assume $k_d = k_x$) because they both have reaction order one and use the same reactant species, even though the products are different.

If one of the reactants for a reaction has no molecules present in the system, then we set the Waiting Time equal to infinity; since we have chosen to implement the algorithm in ANSI C, this is can be easily accomplished as $\frac{10}{0} = \infty$. For higher order reactions, we need to incorporate the other reactants into the calculation of Waiting Time. Following the examples in Table 2.1, a second order reaction (bimolecular) would be calculated in the following way

$$WT_{R_3} = \frac{1}{k_y * |A| * |B|},$$  \hspace{1cm} (2.4)

and a third order reaction (trimolecular) would be

$$WT_{R_4} = \frac{1}{k_z * |A| * |B| * |C|},$$  \hspace{1cm} (2.5)

where $A$, $B$, and $C$ are the reactants required for reactions $R_3$ and $R_4$, $|A|$, $|B|$, and $|C|$ represent the number of molecules present in the system at the moment of Waiting Time calculation, and $k_y$ and $k_z$ are the discrete kinetic constants.

In this way, the initial Waiting Time is calculated for every reaction in the entire system. Now the question remains: how do we efficiently sort the reactions so that we can easily determine which reaction is slotted to occur next? To do this, we will need to build a min-heap (based on reaction Waiting Times), where the top of the heap is the reaction with the smallest Waiting Time – i.e., the next reaction to occur. However, we will not be able to maintain the min-heap, as the Waiting Times
change, in a standard manner. When a rule is applied, multiple nodes can have changes to their Waiting Time, since the multiplicities of the system are changed. Thus, multiple Waiting Times can fail the min-heap property throughout the tree simultaneously after each time step. In order to efficiently evolve the Membrane System, we will need to incorporate some special heap maintenance functions, similar to those proposed by Gibson and Bruck [34] in their modification of the Gillespie Algorithm. This will be discussed further in Section 2.3.

With \( const_{R_i} \) and \( WT_{R_i} \), for each reaction \( R_i \), we have all of the elements for the initialization of our Membrane System. In Section 2.1, we will provide a full description of the NWT algorithm. Then, we will discuss the specifics on our implementation of the Membrane System. The implementation discussion will lead us into our description of the min-heap maintenance. Finally, we will discuss the concept of reaction memory, in order to ensure the proper evolution of the Membrane System.

### 2.2 The Nondeterministic Waiting Time Algorithm

Section 2.1 gave us the data structure for the NWT algorithm. We have a Membrane System, which describes all aspects of the system — e.g., rules, compartments, protein types, numbers of molecules per protein, etc. We discussed the fact that our Membrane System will not evolve in a typical (maximally parallel) manner, because the reactions occur in an asynchronous manner over discrete time intervals of different lengths according to the law of mass action.

Next, we provide a description of the NWT algorithm. The entire source code can be viewed in Appendices B and C. The Membrane System evolves through the
execution of reactions in a Waiting Time-dependent manner until a desired simulation time has been reached. We will now list the Steps for the NWT algorithm.

1. **Build Membrane System:** Import data for Membrane System – alphabet, membrane hierarchy, etc. Convert protein concentrations to molecular multiplicities. Convert kinetic rates to discrete kinetic constants. For each reaction \( R_i \), where \( 1 \leq i \leq m \), we calculate the initial Waiting Time, \( WT_{R_i} \). Choose the desired amount of time for the simulation, \( \tau_{\text{fin}} \). Set current simulation time to zero \( (\tau = 0) \).

2. **Build Heap:** Using the reaction Waiting Times, we build a min-heap of all reactions in the system.

3. **Select Rule:** Choose the reaction with the lowest Waiting Time – the top of the min-heap. Upon selecting the top node, recursively check to see if there are any children nodes sharing the minimum Waiting Time. If such a tie for minimum Waiting Time exists, proceed to step 4. If no tie exists, then proceed to step 5.

4. **Handle Tie:** Check the multiplicities of the reactant species for all tied reactions. If there are enough reactants to satisfy all of the reactions with the minimum Waiting Time, implement all tied reactions. If there are not enough reactants to accommodate all the reactions, use the nondeterministic logic to apply as many rules as possible.

5. **Apply Rule:** Update the multiplicities of the reactant(s) and product(s) for the reaction(s) from step 3. Aggregate the simulation time \( (\tau = \tau + WT_{\text{applied}}) \).

6. **Update Rules:** Recalculate the Waiting Time for all reactions whose reactants
include the products or reactants of the applied reaction(s). That is, we need to see how the multiplicity changes from the applied reaction(s) have affected the Waiting Times for all rules dependent on those proteins with changed multiplicity. For each such reaction compare the new Waiting Time with the existing Waiting Time and keep the smallest of the two (unless the new time is infinity).

7. **Memory Enhancement:** If the recalculation of a reaction’s Waiting Time results in a value of infinity, then we must store the amount of time waited as a percentage ($Mem_{perc}$). If the recalculation of a reaction’s Waiting Time results in a real value and the previous value was infinite, then the Waiting Time will need to be adjusted according to the stored memory percentage.

8. **Heap Maintenance:** Adjust the min-heap, bubbling reaction nodes up or down in order to satisfy the min-heap property, once reaction Waiting Times have been recalculated according to the multiplicity changes. N.B., to accommodate the multiple changes in Waiting Times, we employ nonstandard heap maintenance methods.

9. **Termination:** If $\tau = \tau_{fin}$, then terminate the simulation. Output the multiplicity information for entire simulation. Otherwise, go back to step 3.

The above pseudocode represents a simple explanation of the NWT algorithm; however, we will use the rest of Chapter 2 to elaborate on each step, giving the reader a clear understanding on how to efficiently implement the ideas described in the algorithm. First, we will need to discuss the language and tool used to encode the model for step 1. For this work, we did not seek to develop our own esoteric method.
for encoding the information of a biochemical network. Instead, we have chosen to use a standard language utilized by many systems biology labs across the world.

We feed our simulator with a file encoded in the Systems Biology Markup Language (SBML). SBML is one of the most popular methods to encode biochemical models, developed through a broad international collaborative effort involving the cooperation of many institutions [48]. We chose SBML for its visibility and availability. SBML has in place an extensive emailing group for quick discussions on coding issues and future extensions/developments of the standard. For these reasons, the choice to use SBML instead of similar methods – e.g., CellML – was an easy one.

To generate the SBML files, we use the CellDesigner software [32, 33] which is also the result of a large international collaborative effort but maintained through Keio University. The CellDesigner software provides an easy graphical interface with which to program the models. CellDesigner has many functions, including simulation packages, but we are only concerned with its ability to generate SBML models (pictures and code) through a simple, user-friendly graphical interface.

CellDesigner adds its own extra information to the SBML code, to define the spatial aspects for the graphics – e.g., the locations of membranes and proteins, sizes of proteins, etc. However, the program has the ability to export the model as pure SBML code with none of the extraneous graphical information. There are several different version of SBML code, but we used CellDesigner to generate SBML Level 2 Version 1, the newest standardized SBML format. The SBML file is free of any additions made by CellDesigner. In Figure 2.2, we provide an example of CellDesigner’s graphical representation and SBML code. The Figure represents the
bimolecular reaction which is listed in Table 2.1, and gives us a chance to discuss the various components of the SBML code important for the models in later Chapters.

Within the SBML code, we have everything we need to initialize the Membrane System (step 1 of the NWT algorithm). SBML is a sort of one-size-fits-all language for the description of a biochemical reaction network. As such, we are not interested in using every part of the entire language, for the models present in this work. We will highlight only the parts of the SBML code pertaining to our work, ignoring the components that are of no concern for the models discussed in later Chapters.

The <listOfCompartments> is the first tag of interest to us. It marks the beginning of the model information for populating the compartments, \( L \), and the membrane hierarchy, \( \mu \), of the Membrane System. We associate the compartment \( id \) flag with \( L \) and the outside flag determines where that particular compartment is located with respect to other compartments, \( \mu \). For instance, a nucleus would have outside = "cytoplasm".

The tag <listOfSpecies> contains all of the information used to populate the alphabet, \( \Sigma \), of the Membrane System as well as the multiplicities of each protein per compartment, the sets \( M_1, ..., M_z \) from the Membrane System description in Section 2.1. The flags \( id \) and initialAmount describe the values to be stored in \( \Sigma \) and the \( M_1s \), respectively. The \( ids \) of the proteins are unique, and we use those values instead of name to associate reactants and products with a reaction. As you will see later, we also keep track of name, but we store the name of the protein only to ease the post-simulation analysis. We assume the initialAmount values are always given as concentrations. Therefore, the NWT algorithm needs to convert the protein
concentrations into the numbers of molecules using the basic equation

$$|P| = [P] \cdot N_A,$$  \hspace{1cm} (2.6)

where $[P]$ is the concentration of protein $P$, $|P|$ is the number of molecules, and $N_A$ is Avogadro's constant.

Figure 2.2 Comparison of the graphical representation and SBML code for a model generated using the CellDesigner software.
Next, we need to populate the list of rules \((R_1, \ldots, R_t)\) of the Membrane System. We find this information under the \(<\text{listOfReactions}>\) tag. Under this tag, we find all of the reactions for the entire biochemical network. The description of each individual reaction begins with the \(<\text{reaction}>\) tag. For each reaction, we read the reaction \(id\) flag, which CellDesigner assigns as a unique value per reaction. Two important tags per reaction are the \(<\text{listOfReactants}>\) and the \(<\text{listOfProducts}>\).

In our example, we see \(s1\) and \(s2\) are the two reactants for the bimolecular reaction. Notice, the protein \(id\) is used instead of the \(name\). Following our example, we see there is only one product listed in the SBML code, \(s3\). Finally, we find the kinetic rate, which falls under the \(<\text{parameter}>\) tag. In our implementation, we are not concerned with the rate \(id\), so we ignore this value. We wish only to store the kinetic rate \(value\) flag. Finishing our example reaction, we see the parameter, \(ky\), has value 12.

Using the SBML code, we can populate every aspect of the Membrane System. We have chosen to implement the algorithm in the C programming language. Initially, we programmed the entire algorithm in Java for portability reasons. However, for larger models, like the one described in Chapter 5, we found the speed benefits of C to be necessary in reducing simulation runtime. Also, C gave us the ability to parallelize our simulations via MPI. We mention the fact that we switched from Java to C, because we will begin using the word struct (to facilitate understanding of the source code in the Appendices). However, a Java implementation could be understood by substituting the word struct with object.

All of the elements of the Membrane System can be contained in two arrays of
structs. We refer to one array as the "Alphabet" and the other array as "Reactions". We represent these structs graphically below (Figure 2.3) and invite the interested reader to look at Appendix C for the source code of the two structs. By explicitly explaining these two arrays and the aspects of the structs, we can better understand their relationships and how we have effectively and efficiently implemented the NWT algorithm.

![Diagram of structs](image)

Figure 2.3 The two structs: Protein and Reaction nodes used to build the Alphabet and Rule sets of the Membrane System.

With our discussion of the SBML code, it is already clear what is meant by (a) through (c) of the Protein struct. These components consist of (a) a string, (b) a string, and (c) an integer (the number of molecules must be a whole number). The Compartment and Id components are fixed throughout the entire simulation run. The
Multiplicity component is subject to change throughout the simulation, increasing or
decreasing by one whenever a given protein is the product or reactant of an applied reaction.

Part (d) of the Protein node is the ReactionList array. The ReactionList array contains the indices of all of the reactions for which the protein is a reactant. The purpose of the ReactionList array is $O(1)$ time access to all reactions which use the protein as a reactant. This is convenient for quick recalculation of the Waiting Time of a reaction, after a change in the multiplicity of at least one of its reactants has occurred as the result of the execution of a rule – that is, step 6 of the NWT algorithm.

Finally, component (e) is the Multiplicity array. At each second (easily modified to handle minutes or hours as needed) throughout the entire simulation run, the multiplicities of all proteins are saved to their Multiplicity array. This allows us to circumvent read/write access of the slower memory (IDE/SATA harddrives) in favor of faster memory (RAM). It would be nice to store all the multiplicities in RAM until the simulation has finished, however, the current implementation of the algorithm has the harddrive access after every 40,000 seconds (simulation time). We wanted to take advantage of the built-in declarations of C to ensure compatibility across different systems, compilers, etc. For simulations longer than 40,000 seconds, the Multiplicity array is dumped to a results file every time the array becomes full, then it begins filling again, continuing until the desired simulation time is reached.

Next, we will consider the components of the Reaction structs. Component (a) comes directly from the SBML code and is stored as a string. We discussed earlier (Equations 2.2, 2.3, 2.4, and 2.5), the calculation of the (b) $ConstR$ and the (c) $WT$. 
Both of these values need to be stored with double precision. This is due to the fact that most biochemical reactions take less than one second to occur. Hence, we want to use the largest standard type definition to ensure the largest possible number of decimal places. Values (a) and (b) do not change after initialization. However, as discussed, the Waiting Times are recalculated as the multiplicities of the reactants change.

Components (d) and (e) deal with the memory enhancement of the NWT algorithm. These parts are better left unexplained until the discussion of memory enhancement in Section 2.4. We would like to note that these values are stored with double precision. The HeapIndex (f) is required to handle heap maintenance, which will be explained in Section 2.3.

Finally, the Reactants array and the Products array (components (g) and (h)) contain the indices of the alphabet for the reactants and products of the rules. This allows $O(1)$ time access when updating the multiplicities of the proteins affected by the execution of a given rule, respectively. We allow our algorithm to handle reactions of order no higher than three (trimolecular reactions). Any higher order reaction can be broken down into subsequent smaller order reactions. Hence, the Reactants and Products arrays are not larger than three. This is important for the discussion of runtime later in the Chapter.

This concludes our discussion of step 1 of the NWT algorithm. The information of the Membrane System is completely contained within two arrays of structs: the Alphabet array and the Rules array. Step 2 involves building the min-heap of the array. This step does not require much elaboration, but we will mention the min-heap
property. For any two reactions, $R_1$ and $R_2$, in the heap, if $R_2$ is a child of $R_1$, we must have

$$WT_{R_1} \leq WT_{R_2}.$$  \hspace{1cm} (2.7)

Steps 1 and 2 are each called only once during the simulation. Now we must discuss step 3 of the NWT algorithm: Select Rule. We want to select the reaction with the lowest Waiting Time. Since we have organized our reactions in a min-heap, this step requires only $O(1)$ time to complete. However, we must check to see if any other rules have the same Waiting Time — i.e., reactions attempting to execute at the same exact instant. These competing reactions could potentially be trying to use the same limited reactants. If there are multiple rules slotted to occur at the same moment, then we must ensure there are enough molecules to satisfy all of the reactions. Otherwise, we will need to choose reactions nondeterministically until all available reactants have been exhausted.

We want to create a temporary array to store all the reactions with the minimum Waiting Time. The elements of this array are pointers to nodes in the heap. The first node in the array is the top of the heap. We want to add all reactions to the array with the minimum Waiting Time. To do so, we will recursively check children nodes until we stop finding tied reactions — i.e., reactions attempting to occur simultaneously. The recursive method is presented below:

1: \textbf{CheckTie}($WT$, \textit{parent})

2: \hspace{1cm} \textbf{IF} leftchild exists

3: \hspace{2cm} \textbf{IF} $|parent(WT) - leftchild(WT)| < \epsilon$

4: \hspace{3cm} add leftchild pointer to ties array
5:       CheckTie(WT, leftchild)

6:       IF rightchild exists

7:       IF |parent(WT) − rightchild(WT)| < ε

8:       add rightchild pointer to ties array

9:       CheckTie(WT, rightchild)

Consider lines 3 and 7. Ideally, we would like to check for equality between the 
parent and child. However, due to the way the C programming language handles 
double precision, we are not allowed to make exact comparison, e.g., \textit{parent}(WT) = 
\textit{leftchild}(WT). Instead, we use an error tolerance, \(\epsilon\), to declare two doubles are 
'equal'. In our simulations, we have defined the error tolerance as \(10^{-13}\), as it worked 
well during simulation testing.

If the two Waiting Times are within \(\epsilon\) of each other, then they are considered 
to be equal. The child is added to the array of ties, and the CheckTie method is 
recursively called with \textit{child} as the passed parameter. This continues until no more 
children have the minimum Waiting Time. Once the method (and any recursive calls 
to the method) terminates, then we have a complete ties array. If there is only one 
element in the ties array, then there is no tie, and we can move on to step 5 and 
apply only the one rule. Otherwise, we must apply as many rules as possible in a 
nondeterministic manner.

To nondeterministically apply rules, we define a random number generator using 
the system clock as the random seed. We randomly generate numbers between 0 and 
the end of the ties array. Using this randomly chosen index, we check if sufficient 
reactants exist to implement the reaction. If there are sufficient reactants, we apply
the reaction – i.e., we increase the multiplicity of the product(s) by one and decrease the multiplicity of the reactant(s) by one. If there are insufficient reactants, we skip the reaction, and no multiplicity changes occur for the reaction. In either case, the reaction is removed from the ties array, and the process continues until the ties array is empty. This completes step 4 and step 5. Recall, in the case of only one reaction, we skip step 4 and apply just the one reaction in step 5. In either case, we are ready to move on to step 6: Update Rules.

Step 6 of the algorithm requires access to component (g) and (h) of the Reaction struct and component (d) of the Protein struct. For each reaction applied in step 5, we must recalculate the Waiting Time of the applied reaction and the Waiting Time of every reaction affected by the multiplicity changes. We must discuss step 6 within the context of the heap maintenance. Hence, we will continue the discussion of step 6 in Section 2.3, which will also be a discussion of step 8, Heap Maintenance. As you will see in the next two Sections, steps 6, 7, and 8 are all intertwined. But, the discussion on step 7 is left for Section 2.4.

2.3 Maintaining the Min-heap

As we stated earlier, we are building a min-heap from our Rules array with the bottom-up method. However, the maintenance of the heap is accomplished in a nonstandard way. Standard methods for heap maintenance involve selecting the top node and removing it from the heap entirely. Meanwhile, new nodes are added to the bottom and bubbled up as necessary. There are a couple of reasons why we do not want to remove the nodes of applied rules from the heap and add them to the
bottom.

For one, the number of reactions will not grow or shrink during a simulation run. Hence, we do not need to remove or add reactions to the heap once it has been initialized. Second, it is most often the case that, once a reaction is applied, its new Waiting Time is very close to the previous value. Therefore, the top node will most likely be located near the top of the heap once the heap is resorted. Hence, popping the top node and adding it to the bottom will often result in the node being bubbled back up to near the top of the tree (a waste of computer clock cycles, especially for a significantly large numbers of reactions).

As we said at the end of Section 2.1, step 6 and step 8 are must be handled concurrently. We will provide the pseudocode for the two steps below, then we will clarify their function with an example.

1: \textbf{UpdateRules}()

2: \textbf{FOR} each applied $R_i$

3: \hspace{1em} Recalculate $WT_{R_i}$

4: \hspace{1em} FixHeap($R_i$ node)

5: \textbf{FOR} each reactant $X$ of $R_i$

6: \hspace{1em} \textbf{FOR} each reaction $R_x$ requiring $X$ as a reactant

7: \hspace{2em} Recalculate $WT_{R_x}$

8: \hspace{2em} FixHeap($R_x$ node)

9: \textbf{FOR} each product $Y$ of $R$

10: \hspace{1em} \textbf{FOR} each reaction $R_y$ requiring $Y$ as a reactant

11: \hspace{2em} Recalculate $WT_{R_y}$
12: FixHeap($R_y node$)  

13: FixHeap ($node$)  

14: IF ($node$ has a parent)  

15: IF (parent(WT) > node(WT))  

16: Swap the parent with node  

17: FixHeap(node)  

18: IF ($node$ has leftchild or rightchild)  

19: IF (node(WT) > leftchild(WT) or node(WT) > rightchild(WT))  

20: Swap node with the child with that smallest WT  

21: FixHeap(node)  

To understand the relationship between steps 6 and 8 of the NWT algorithm, we will follow the maintenance of the heap with an example tree. In Figure 2.4, we provide a tree with twelve reactions (A, B, C, D, E, F, G, H, I, J, K, and L). The Waiting Times for each reaction are also provided (.1, .3, .4, .6, .7, .8, .9, and $\infty$) in the heap. We see the tree is indeed a min-heap. The reaction with the smallest Waiting Time is reaction E ($WT_E = 0.1$).
There are a few important assumptions we must state for the example problem. First, we assume the algorithm has just executed reaction E. In other words, reaction E was selected in step 3, and the multiplicities of the reactants and products of reaction E have been updated to accommodate the execution of the rule (step 5). But, we have not yet recalculated the Waiting Time for reaction E. So, in our discussion of the example, we will assume that we are starting from step 6 of the algorithm, which begins at line 1 of the above pseudocode after a call to the UpdateRules method. We have also marked in Figure 2.4 all of the nodes requiring a recalculation of Waiting Time after the multiplicity changes have occurred as a result of the execution of reaction E. These affected rules are all marked in gray.

Besides recalculating the Waiting Time for the applied reaction E, we assume that the multiplicity changes brought on by the execution of reaction E will affect the Waiting Times of reactions J and I. Finally, we notice that before any Waiting Time is recalculated, the tree satisfies the min-heap. This is an important property for understanding the implementation of steps 6 and 8. It ensures that after each Waiting Time recalculation, the min-heap property is fully satisfied, before moving on to the next recalculation.

In this particular example, there is no tie for minimum Waiting Time. Hence, the first FOR loop, line 2, runs only a single iteration. The Waiting Time of reaction E is recalculated and the result is shown in Figure 2.5. We see that reaction E now has a Waiting Time of $\infty$. From this, we know that one of its reactant species was completely exhausted by the execution of reaction E. Also, we see that the tree is now in violation of the min-heap property. The FixHeap method is called.
Figure 2.5 Recalculation of the Waiting Time for reaction E.

The FixHeap method is called on the node representing reaction E. Figure 2.6 illustrates the steps required to adjust the tree. First, the node (reaction E) checks for a parent, line 14. It has none, so lines 15-17 are ignored. However, reaction E does have a left child and a right child (line 18). Furthermore, we see that the Waiting Times for the two children (reactions F and J) are both smaller than the Waiting Time for reaction E, satisfying the IF statement on line 19. Then, according to line 20, we must swap reaction E with the child with the smallest Waiting Time (reaction F). We see the change in Figure 2.6(i). After swapping the two nodes, we call the FixHeap method again, passing reaction E (in its new position) as the parameter.

Continuing with Figure 2.6(i), we consider the FixHeap method on reaction E. The reaction now has a parent (reaction F), satisfying line 14. However, the IF statement on line 15 fails, since the Waiting Time of reaction E is greater than the Waiting Time of reaction F. Once again, the left and right children of reaction E both have smaller Waiting Times, so we swap reaction E with the smaller of the two (reaction B). We see the result of this transition in Figure 2.6(ii). With line 21, we make another call to the FixHeap method with reaction E.
On the next call to the FixHeap method, we see reaction E will move down one more level. Reaction E is compared to reaction K and reaction A. The Waiting Time of reaction E is larger than the Waiting Time of reaction K but not larger than the Waiting Time of reaction A. Regardless, the IF statement on line 19 is satisfied, and reaction E is swapped with reaction K according to line 20. A final call to the FixHeap method is made on reaction E. However, with the final call, lines 15 and 19 cannot be satisfied. Hence, the method terminates. We see in Figure 2.6(iii) that the min-heap...
property is satisfied for every node in the tree. However, we need to continue the method UpdateRules, since we still must cope with the affected rules (reaction I and reaction J).

Now that reaction E has been recalculated and we have a min-heap, we can continue the pseudocode at line 5. For lines 5-12, the NWT algorithm is able to recalculate each rule affected by the applied rule, and update the heap to accommodate the changes made to the Waiting Times of the affected rules (I and J). Without loss of generality, we can assume that reaction I is recalculated before reaction J. After explaining the heap maintenance for reactions I and J, we will provide an explanation on the irrelevance of choosing I before J. For now, we continue by assuming that the recalculation of the Waiting Time for reaction I is smaller than the previous value. Hence, the WT is changed and the heap now violates the min-heap property at node I. We see the the violation of the min-heap property in Figure 2.7.

![Diagram of a tree with nodes labeled F, B, J, H, K, G, L, C, D, E, A, and 1.3.

Figure 2.7 Recalculating the Waiting Time for reaction I.

After the Waiting Time for reaction I is recalculated (line 11), we call the FixHeap method for node I (line 12). We see reaction I has a parent, and its parent node (reaction G) has a Waiting Time greater than the Waiting Time of reaction I. Therefore,
the two nodes are swapped (line 16). The FixHeap method is called on reaction I (line 17) from its new position (see Figure 2.8(i)). With this call to FixHeap, we have yet to complete lines 18-21 in the original FixHeap call. For this first call to FixHeap, this will result in no changes, since everything below reaction I (in its new position) satisfies the min-heap property.

It is clear from Figure 2.8(i) that reaction I will need to move up one more position, switching places with reaction J. Once this happens, the min-heap property is satisfied and all of the calls to FixHeap will result in no additional changes before terminating. We note that in Figure 2.8(ii), we indeed have a min-heap. Now we have just one final reaction to handle (reaction J). We recalculate the Waiting Time of reaction J and find that it is now smaller than the previous value (see Figure 2.9(i)).
Following the pseudocode, we see that reaction J will be compared to its parent node, reaction I. Since the Waiting Time of reaction J is lower than the Waiting Time of reaction I, the two nodes are swapped (Figure 2.9(ii)) and FixHeap is called on reaction J (in its new position). Additionally, the Waiting Time of reaction J is lower than the Waiting Time of reaction F. The two nodes are swapped and the result is a tree that fully satisfies the min-heap property (Figure 2.9(iii)). This concludes the example problem. Each node was recalculated and adjusted before the algorithm
continues on to the next affected reaction.

Now, we must comment on the irrelevance of checking reaction I before reaction J. The way we have implemented steps 6 and 8 allows us to ignore the order to check the affected reactions. Prior to each Waiting Time calculation, the tree satisfies the min-heap structure. Consider the recalculation of a reaction’s Waiting Time. One of three things can happen: (a) Waiting Time increase (to infinity), (b) Waiting Time decrease, or (c) Waiting Time does not change. In the case of (c), the reaction will fail for lines 15 and 19. In other words, if there are no changes to the Waiting Time, then there need be no changes to the heap. For (a), the reaction may need to move down in the tree, but it will certainly not move up. Therefore, only lines 18-21 are required for (a). Finally, if the Waiting Time decreases, then only lines 14-17 are relevant – i.e., the smaller Waiting Time may cause the reaction to move up the tree.

Regardless, once the FixHeap method has been called (along with subsequent recursive calls) on a node satisfying one of the three properties above (a) through (c), the tree will satisfy the min-heap property. We do this for each reaction affected by the execution of the reaction with the minimal Waiting Time (step 3), and we see our algorithm efficiently sorts the reactions.

The implementation of our heap yielded a massive performance increase over the previous algorithm from [17]. While incorporating the heap structure we not only increased the sorting performance, we were able to eliminate an extraneous FOR loop (running for every reaction in the tree) used to put Waiting Times in the context of simulation times. Our previous simulator had a runtime of $O(n^2 \log n)$. To be able to give the complexity of the algorithm proposed and show that it is indeed efficient, we
need to make several assumptions which are (usually) valid for the signaling cascades:

1. each reaction involves at most 5 different species of molecules;

2. there are a bounded number of reactions having the same reactant (usually 3, at most 5);

3. there are not many reactions happening at the same time (due to the differences in molecule multiplicities and reaction rates).

From 1, 2 and 3 we can now state that our new algorithm has a runtime of $O(n \log n)$ with respect to the number of reactions simulated.

This concludes our discussion of steps 6 and 8. We skipped step 7, but we will now explain it in Section 2.4. Technically, steps 6, 7 and 8 all happen at the same time. We see the interplay with steps 6 and 8 above, and step 7 merely factors into the recalculations of the Waiting Times (lines 3, 7, and 11 of the above pseudocode).

### 2.4 Memory Enhancement

There are often situations in biochemical networks, where one (or more) protein(s) ($p_i \in \Sigma$) is a reactant for two or more reactions of different kinetic rates (fast vs. slow). In order to explain our memory enhancement, we will consider an example system involving only three reactions ($R_1$, $R_2$ and $R_3$) acting on four proteins ($A$, $B$, $C$, and $D$). The model is described in Table 2.2.

When described as a system of ordinary differential equations, the biochemical network is given in Equation 2.8.
Table 2.2 An example system to illustrate memory enhancement.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Rate Constant</th>
<th>Initial Molecules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$: $A \rightarrow C$</td>
<td>$k_1$ (slow)</td>
<td>$A = 1$</td>
</tr>
<tr>
<td>$R_2$: $A \rightarrow B$</td>
<td>$k_2$ (fast)</td>
<td>$B = 0$</td>
</tr>
<tr>
<td>$R_3$: $D \rightarrow D + A$</td>
<td>$k_3$</td>
<td>$C = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$D = 1$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\frac{d[A]}{dt} & = -k_1[A] - k_2[A] + k_3[D] \\
\frac{d[B]}{dt} & = k_2[A] \\
\frac{d[C]}{dt} & = k_1[A] \\
\frac{d[D]}{dt} & = 0
\end{align*}
\]

(2.8)

We specifically designed the system above to highlight the effects of the memory enhancement, and we will compare our refined algorithm with solutions to the system of ordinary differential equations described in Equation 2.8. A model similar to this one could be used to investigate the dynamics of human immunodeficiency type 1 (HIV-1) Tat protein, since it is initially transcribed at very low numbers [57]. Once Tat is assembled in the cytosol, it can be exocytised or translocated to the nucleus [109]. When Tat is translocated to the nucleus it can begin upregulating HIV-1 proteins (including itself). Since the downstream effects of Tat translocation to the nucleus has profound impacts on the cell (causing upregulation of the HIV-1 proteins), a discrete and nondeterministic approach is necessary to follow the dynamics of the low levels of Tat proteins [124].

In the system, molecules of $A$ are formed from molecules of $D$. This reaction can basically be viewed as a combined transcription and translation rule with $D$ being the
gene and $A$ being the protein encoded by the gene. Once a molecule of $A$ is formed, it has the option doing one of two things: (i) turning into a molecule of $B$ at rate $k_2$ or (ii) turning into a molecule of $C$ at a rate $k_1$. If we consider the species $A$ as being analogous to HIV-1 Tat protein, then $A \rightarrow B$ could be translocation to the nucleus and $A \rightarrow C$ could be translocation to the extracellular environment.

Next, we will look at two cases for the model described in Table 2.2 and discuss the memory enhancement. The cases vary by choices of the kinetic rates. The first case shows that the memory enhancement can produce the same results as the deterministic differential equations approach. For the second case, we will show how the technique can produce different results, illustrating that the ability of the NWT algorithm to explore nondeterminism of molecular signaling cascades. There are no modifications to the NWT algorithm between the two cases. The only difference is the kinetic constants for initialization of the model.

### 2.5 Case 1: Deterministic Memory Enhancement

For the first case, we let $k_1 = 10$, $k_2 = 4$, and $k_3 = 5$. The results of a simulation using the NWT algorithm plotted against the solution of the system of ordinary differential equations is shown in Figure 2.10. The graph shows the accumulation of $C$ molecules throughout a 10 second run. The bars of the graph are the discrete simulation results using the NWT algorithm, and the black line is the solution to the system of ordinary differential equations. With the choices of the kinetic values, there are no nondeterministic decisions for the entire length of the NWT simulation. So,
there is no surprise that the NWT algorithm shows the same increase in C molecules as the solution to the system of ordinary differential equations.

At initialization \((t = 0)\), there is exactly one molecule of \(D\) and one molecule of \(A\). Therefore, from Equation 2.3, we see that all three reactions have real (finite) waiting times when the simulation begins. Furthermore, we have \(WT_{R_1} = 0.25\), \(WT_{R_2} = 0.1\) and \(WT_{R_3} = 0.2\). Using these \(WTs\), we are able to build a min-heap where the top node is \(R_2\), since it has the smallest Waiting Time.

The first reaction to occur is \(R_2\), which immediately exhausts the system’s supply of \(A\) molecules, yields one molecule of \(B\) and a simulation time of \(t = 0.1\). The rules affected by the applied rule must be recalculated; the Waiting Times are now \(WT_{R_1} = WT_{R_2} = \infty\). Since \(R_3\) does not require a molecule of \(A\), \(WT_{R_3}\) is left unchanged after the first reaction is executed. Also, upon readjusting the min-heap, \(R_3\) is now at the top since it has the smallest value for \(WT\).

The next reaction to be applied is \(R_3\), which gives us a new molecule of \(A\) and a simulation time of \(t = 0.2\). This is where the memory enhancement plays a role. In the first step, reaction \(R_2\) used up all of the molecules of \(A\). However, when this happened, \(R_1\) had already waited for 0.1 seconds—the amount of time a molecule of \(A\) was in the system. The memory enhancement allows the simulator to keep track of the percentage of time waited. In other words, \(R_1\) waited for 0.1 seconds out of its required 0.25 seconds, which means it has waited 40% of its Waiting Time. If we allow the algorithm to keep track of this percentage, then, when a new molecule of \(A\) is formed, we can recalculate the \(WT\) for reaction \(R_1\), using the percentage to adjust its Waiting Time accordingly. That is, after \(R_3\) is applied in step two and we have a
new molecule of \( A \), we recalculate \( WT_{R_i} \) using Equation 2.3, but we take 60% of this number – the percentage of time left to wait.

Figure 2.10 The number of \( C \) molecules for a simulation of \( t = 10 \). The NWT algorithm results are represented with bars while the solution to the system of ordinary differential equations is the solid line.

To recap, the first reaction applied is \( R_2 \). The Waiting Times for reaction \( R_2 \) and \( R_1 \) are both recalculated as infinity, since \( R_2 \) used all the molecules of \( A \) present in the system. The memory enhancement allows \( R_1 \) to store the percentage of time it has left to wait (60%) when \( A \) is exhausted. Next, \( R_3 \) is the second reaction to be applied. When this occurs, there is a new molecule of \( A \) in the system, which means \( R_2 \) and \( R_1 \) need to be recalculated. The Waiting Time of \( R_2 \) is calculated as 0.1, but
the Waiting Time of $R_1$ is recalculated as 0.15. This number stems from the equation

$$WT_{R_3} = Mem \frac{1}{k_3 * |A|}$$

where $Mem$ is the percentage of time left to wait (60% in the example above). We will reiterate the memory enhancement calculations in the second case study with different kinetics.

In this case, the solutions to the system of ordinary differential equations and the results from the NWT algorithm agree. In the next case, we will explain how the nondeterminism of the NWT algorithm can lead to results different than continuous, deterministic solutions to system of ordinary differential equations.

### 2.6 Case 2: Nondeterministic Memory Enhancement

We will now modify the kinetic constants to highlight the effects of the nondeterministic component of the NWT algorithm in conjunction with the memory enhancement. Although the kinetics of our sample system have been deliberately chosen to highlight the nondeterministic effects, we will later show, in Chapter 3, how our nondeterministic logic can have similar implications in a known model, comparable to the Gillespie Algorithm – deviating from deterministic simulations – but at a considerably reduced computational cost.

For our next simulations, we assume $k_1 = 0.1$, $k_2 = 1.0$, and $k_3 = 0.5$. The initial Waiting Times are initialized as $WT_{R_1} = 10$, $WT_{R_2} = 1$, and $WT_{R_3} = 2$. In Figure 2.11, we see the accumulation of $B$ and $C$ molecules. The results of the ODE-based simulation are visibly different than the results of the NWT algorithm. The reasons
for the differences are the nondeterministic decisions on reaction competition for $A$ molecules.

Figure 2.11 Results of the memory enhancement simulation for the ordinary differential equations and the NWT algorithm. (a) molecules of $B$ and (b) molecules of $C$.

In both graphs we see the results of the solution to the system of ordinary differential equations (straight black line) shown with many runs with the NWT algorithm. In the two graphs, we have (a) the number of molecules of $B$ and (b) the number of
molecules of $C$ are shown. Molecules of $B$ and $C$ both come from $A$ molecules. However, the reaction for $B$ is faster than the reaction for $C$. In the solution to the system of ordinary differential equations, a molecule of $A$ can be used to partially satisfy $B$ and $C$. Since our NWT algorithm is discrete, the molecules are nondeterministically chosen to satisfy one or the other – similar to cellular processes. The reaction changing $A$ into $C$ ’remembers’ how long it has waited, and uses this information the next time a molecule of $A$ is ready.

Based on the initialized Waiting Times, the first reaction to be applied is $R_2$. After $R_2$ is applied, the simulation time is aggregated ($t = 1$) and there are no more molecules of $A$ present in the system. Similar to the previous case study, the Waiting Times for $R_2$ and $R_1$ are both set to infinity (no reactant molecules available). However, we store the percentage of time waited ($Mem$) for the slow reaction $R_1$ – in this case, $Mem_{R_1} = 90\%$. Since $R_3$ is unaffected by the execution of the first reaction, it now has the minimum waiting time. The next rule to be applied is $R_3$. The simulation time is adjusted ($t = 2$), and we now have a new molecule of $A$. With our new $A$ molecule available, we must recalculate the Waiting Times for $R_1$ and $R_2$.

Using the $Mem_{R_1}$, we can recalculate the Waiting Time for reaction $R_1$ and use the fact that it has already waited 10\% of its Waiting Time. Therefore, when a new molecule of $A$ is formed two seconds into the run, we recalculate the $WT_{R_1}$ using Equation 2.9. In our case, we have $WT_{R_1} = 9$ and $WT_{R_2} = 1$.

In a strictly deterministic sense, our algorithm is capable of generating equivalent results to an ordinary differential equations model (see case 1 above). But, with the nondeterminism of our algorithm, the memory enhancement can lead to different
results. Continuing the calculations for the simulation, we skip ahead to a future event \((t = 18)\). Up until this point, we have been creating molecules of \(A\), and every single one of them has been deterministically chosen to change into molecule \(B\) via reaction \(R_2\). But, at \(t = 18\), a molecule of \(A\) has been created, and the Waiting Times of reaction \(R_1\) and \(R_2\) are equal \(WT_{R_1} = WT_{R_2} = 1\). The reason for this is that we have \(Mem_{R_i} = 10\%\). In other words, \(R_1\) and \(R_2\) are competing to use the same single molecule of \(A\) to form a \(C\) and \(B\) molecule, resp. The solution to the system of ordinary differential equations has no issue at this timepoint, because, whereas our simulator represents molecules of \(A\) discretely and has only allowed reaction \(R_2\) to occur so far, the differential equations simulation is sending a fraction of each \(A\) to form a fraction of \(B\) and \(C\). This is merely a consequence to the way solutions to systems ordinary systems of differential equations behave.

Our algorithm faces the question: at \(t = 18\) should the \(A\) molecule be allowed to satisfy \(R_1\) or \(R_2\)? The algorithm answers the question by making a nondeterministic choice between \(R_1\) and \(R_2\) (step 3 of the NWT algorithm). If \(R_1\) is chosen, then it is applied, and our results stay with the ordinary differential equations results (up to \(t = 19\)). Remember, the ordinary differential equations have been slowly and continuously aggregating the \(C\) molecules throughout to reach one full molecule of \(C\) at time \(t = 19\). However, if \(R_2\) is chosen, then our solution diverges from the previous solution. When the effects of the nondeterministic decisions are aggregated over 1000 seconds, we see the different results obtained from the NWT algorithm (Figure 2.11).

With this example system, we show how reaction memory can affect results. The memory enhancement is designed to give the NWT algorithm results more like solu-
tions to ordinary differential equations in a strictly deterministic sense. However, as a consequence to the reaction memory, we have possible divergence in situations of very low molecular multiplicity, due to the nondeterministic component of the NWT algorithm.

2.7 Summary

In Chapter 2, we have provided all of the information necessary to understand and implement the Nondeterministic Waiting Time algorithm. We introduced Membrane Systems along with a discussion on why this particular paradigm is useful to apply to modeling biochemical reaction networks. We provided pseudocode for the NWT algorithm, and the interested reader can look at the Appendices (B and C) for our C encoding. We also offered an explanation on the specific improvements over previous work from the modeling group of the advisor for this dissertation, Dr. Andrei Păun.

For the rest of this dissertation, we will be exploring models reported in the literature. Then, in Chapter 5, we provide the framework for a new model, investigating the effects of the human immunodeficiency virus on T cells. But first, in Chapter 3 will explore two popular models, to emphasize the NWT algorithm’s unique perspective on biochemical network simulation.
CHAPTER 3

EXPLORING SOME COMMON MODELS

In Chapter 2, we provided the pseudocode for the NWT algorithm, as well as an in-depth look at its implementation, some results of example systems, and a discussion on efficiency. Now we will use the NWT algorithm to explore some popular models in computational and systems biology. We will compare the results obtained from the NWT algorithm with results obtained from systems of ordinary differential equations and the Gillespie Algorithm. The primary goal of Chapter 3 is to illustrate the ability of the NWT algorithm to utilize deterministic and nondeterministic decision-making to produce results similar to ordinary differential equations for some models and results similar to the Gillespie Algorithm for other systems.

3.1 Lotka-Volterra Predator-Prey Model

The Lotka-Volterra predator-prey model depicts the interactions of two species. It was developed independently by both individuals [94]. We will briefly discuss the motivation behind the model. First, we assume a prey population, \( P_1(t) \), and predator population, \( P_2(t) \), which represents the number of each species at time \( t \). We can think of these two populations as lion and zebra, shark and fish food (the particular study of interest to Volterra), etc. In order to model population dynamics, we need to associate to each species a reproductive rate (\( r_{P_1} \) and \( r_{P_2} \)).
We assume that without any predators, the prey population will follow the Malthusian model [94]. However, predators will negatively affect the reproduction rate. Hence, we have \( r_{P_1} = a - b \cdot P_2 \). We also assume that without any prey, the predator will follow the Malthusian model (but with a negative reproductive rate). Furthermore, the prey will increase the production rate of the predators. So, we have \( r_{P_2} = -c + d \cdot P_1 \).

Now we can formalize the predator-prey model as a pair of first-order, nonlinear, differential equations

\[
\begin{align*}
\frac{dP_1}{dt} &= P_1 \cdot (a - b \cdot P_2) \\
\frac{dP_2}{dt} &= -P_2 \cdot (c - d \cdot P_1)
\end{align*}
\]

In Figure 3.1, we see a picture of the predator-prey model. The picture (as well as the SBML code for the model) was generated using CellDesigner [32, 33]. The Lotka-Volterra model involves two interacting species. Prey species are born at a rate \( a \) and are consumed at a rate \( b \) by the predator species. The predator species are born at a rate of \( d \) if there is available food (prey). The way the system is designed, an increase in prey leads to an increase in predator, and an increase in predator leads to a decrease in prey. Total annihilation of prey leads to total extinction of predator, since the food supply of the predator will be exhausted.

We used three different simulation techniques to model the reactions described in the Lotka-Volterra model: a system of ordinary differential equations (Equation 3.1), the Gillespie Algorithm, and our NWT algorithm. The system of ordinary differential equations was solved in MATLAB, while the other two algorithms were both coded
in C. Below we see the results of the three simulations in Figure 3.2 for one hundred seconds.

![Diagram of the Lotka-Volterra model](image)

Figure 3.1 The Lotka-Volterra model.

The solution to the ordinary differential equations in Equation 3.1 shows consistent oscillations throughout the entire simulation run. The NWT shows dampened oscillations over time. The Gillespie Algorithm has difficulties producing the oscillations, due to the stochasticity of the algorithm. In this case, our NWT algorithm runs deterministically. The system is small enough and the dynamics are such that the NWT makes no nondeterministic decisions due to reaction competition. If we expand the results of the solution to system of ordinary differential equations and the NWT algorithm, we see further decline in the amplitude for the NWT algorithm. In Figure 3.3, we expand the simulation run for a total of five hundred seconds. The
results for the solution to the system of ordinary differential equations and the NWT algorithm simulation are provided.

Figure 3.2 Results of three simulation techniques for the Lotka-Volterra model (up to 100 seconds). (a) solution to ordinary differential equations, (b) the NWT algorithm, and (c) the Gillespie Algorithm.

We modeled this classic system to illustrate the differences in the results of our simulation technique compared to the solution of the system of ordinary differential equations, the NWT algorithm, and the Gillespie Algorithm simulations. Our system was able to exhibit oscillatory behavior, albeit the oscillations are damped. However, as you can see in Figure 3.3, the oscillations persist with the NWT algorithm (and the ordinary differential equations). Yet, the Gillespie Algorithm will always reach a steady state, whereby the predator and prey species will eventually completely disappear. Since there are no nondeterministic decisions made during the run, we
can only attribute the dampened oscillations to the fact that the system is discrete. We will next discuss a circadian rhythm model, which will illustrate how our algorithm can produce Gillespie-like results, even though we have a reduced complexity.

Figure 3.3 Results of the two simulation techniques for the Lotka-Volterra model (up to 500 seconds). (a) solution to ordinary differential equations and (b) the NWT algorithm.

3.2 Circadian Rhythm

Circadian rhythm models are often explored in nature. These act as internal clocks which allow organisms to anticipate daily changes in the environment [6] – for instance, when to hunt for food, when to rest, etc. Yet, at the level of cellular
biochemistry, circadian rhythms have also been reported [25]. Biological systems run by internal clocks – that is, certain proteins are created at certain parts of the day. Therefore, simulating circadian rhythm models is important in understanding the way DNA is interpreted and pre-existing proteins waiting to be activated are used by the body for daily survival [6].

We have chosen to model the circadian rhythm model described in [118]. The system describes an activator and a repressor gene (A and R). These genes are transcribed into mRNA, which leads into the translation of the proteins. The activator A binds to the promoters for A and R and increases the transcription rate. The system of ordinary differential equations described in [118] showed that intrinsic biochemical noise enhanced the oscillations. In Equation 3.2, we see the system of ordinary differential equations for the model.

\[
\begin{align*}
\frac{dD_A}{dt} &= \theta_A * D'_A - \gamma_A * D_A * A \\
\frac{dD_R}{dt} &= \theta_R * D'_R - \gamma_R * D_R * A \\
\frac{dD'_A}{dt} &= \gamma_A * D'_R * A - \theta_A * D'_A \\
\frac{dD'_R}{dt} &= \gamma_R * D_R * A - \theta_R * D'_R \\
\frac{dD_{MA}}{dt} &= \alpha'_A * D'_A + \alpha_A * D_A - \delta_{MA} * M_A \\
\frac{dA}{dt} &= \beta_A * M_A + \theta_A * D'_A + \theta_R * D'_R - A * (\gamma_A * D_A + \gamma_R * D_R + \gamma_C * R + \delta_A) \\
\frac{dM_R}{dt} &= \alpha'_R * D'_R + \alpha_R * D_R - \delta_{MR} * M_R \\
\frac{dR}{dt} &= \beta_R * M_R - \gamma_C * A * R + \delta_A * C - \delta_R * R \\
\frac{dC}{dt} &= \gamma_C * A * R - \delta_A * C
\end{align*}
\]
where \( A \) and \( R \) represent the number of activator and repressor proteins, \( D'_A \) and \( D_A \) represent the number of activator genes with or without binding to \( A \), \( D'_R \) and \( D_R \) represent the number of repressor genes with or without binding to \( R \), \( M_A \) and \( M_R \) represent mRNA molecules of \( A \) and \( R \), and \( C \) represent the corresponding inactivated complex formed by \( A \) and \( R \).

Figure 3.4 Picture of the circadian rhythm model.

Deterministic modeling techniques, like the solution to the systems of ordinary differential equations, for biochemical interactions fail to produce the oscillations of a circadian rhythm model. However, the stochastic noise from a Gillespie-based approach leads to repeated oscillations throughout an entire run. Our NWT algorithm can produce results similar to the Gillespie algorithm – genetic oscillations – but at a considerably reduced computational cost.
The results for the simulation of the circadian rhythm model are shown in Figure 3.5. We present the results from Gillespie's Algorithm, the solution of the system of ordinary differential equations (Equation 3.2), and our NWT algorithm. The NWT algorithm is able to reproduce the oscillations for the perturbed model, as is the case with the Gillespie approach [118]. Similar to Gillespie, the NWT shows some variability in both the amplitude – numbers of molecules – and the periodicity of oscillations.

![Figure 3.5 The results for the circadian rhythm model: (a) the NWT algorithm and (b) the Gillespie Algorithm. Both algorithms are plotted against the solution to the system of ordinary differential equations.](image)
The authors in [118] showed that parameter values can have a profound impact on oscillations. By reduction of the kinetic rate governing $R$ degradation, the deterministic results produce a single peak followed by a steady state, while a stochastic simulation remains oscillating. Our NWT algorithm also produces oscillations instead of a steady state, but at a reduced computational cost from the Gillespie Algorithm approach. This is the benefit of modeling with the NWT algorithm instead of the Gillespie Algorithm.

For our simulation to produce oscillations comparable to the Gillespie Algorithm, we require only 50 random numbers to be generated. This stems from the fact that the NWT algorithm relies on deterministic kinetics for the majority of reactions, but when reactants are limited and competition for reactants exists, nondeterministic decisions drive a variable response from the competing reactions.

### 3.3 Summary

In Chapter 3, we have provided two popular models. The models were used to illustrate how our technique compares to the Gillespie Algorithm in situations of low molecular multiplicity. For the circadian rhythm model, we were able to show oscillations where the ordinary differential equations failed to do so. In modeling the Lotka-Volterra model, we showed (dampened) oscillations where the Gillespie Algorithm failed (see Figures 3.2 and 3.3). This aligns with our claim in Chapter 2 that the NWT algorithm lies between the Gillespie Algorithm and solutions to systems of ordinary differential equations. In the Chapter 4, we will apply our algorithm to a model for Fas-mediated apoptosis.
CHAPTER 4

MODELING FAS-INDUCED APOPTOSIS

The term apoptosis was coined in the classic work of Kerr, Wyllie and Currie [58]. Within that paper, the authors described apoptosis as a form of cell death distinct from necrosis, citing a lack of inflammation of the tissue among other differences. Apoptosis is often used synonymously with programmed cell death, emphasizing a cell’s inherent genetic predisposition for death. In other words, the cell’s genome contains the instructions for the cell’s self-destruction. However, we note that there are other accepted forms of programmed cell death described in the literature [24, 31, 112]. These types of cell death – for example, aponecrosis – are characterized as sharing some of the characteristics of necrosis and/or apoptosis. Essentially, apoptosis is a clean and effective method for the elimination of unwanted or damaged cells within the organism. It is sometimes called cellular suicide; the cell receives a message to die and, based on its biochemical composition at the time the death message is received, the cell ‘decides’ whether to live or die. Furthermore, there are additional situations where apoptosis is not a programmed cell death, such as the case with some cancers and other disorders [42, 72, 82, 84].

Thus, cellular apoptosis is an important process in biological systems. Through-
out the entire lifespan of an organism, death by apoptosis is essential for maintaining cellular homeostasis. Indeed, in [103], the authors estimate that a typical human being will produce ten billion cells daily from stem cells to replace the ones dying from apoptosis. There are some especially vital roles for apoptosis in early embryonic development. For example, a developing human initially overproduces the cells of the nervous and immune systems; however, those cells lacking synaptic connections (neurons) or functional antibodies (B cells and T cells) are subject to death via apoptosis (reviewed in [84] and [98], respectively). Aside from a critical early developmental role, apoptosis has also been related to aging effects. For instance, one theory on aging, involving the oxidative stresses on the mitochondria induced by harmful free radicals, illustrates age-related apoptotic cell death [45, 99].

Under the microscope, apoptosis manifests through a series of physiological changes. The cell shrinks and condenses, while the cytoskeleton collapses and the nuclear envelop disassembles. Pyknosis (the condensation of the chromatin) occurs, which is followed by cell surface blebbing, leading to the formation of apoptotic bodies containing all the intracellular material. These self-contained, apoptotic bodies can then be phagocytised by macrophages, parenchymal cells, or neoplastic cells. In this way, organelle integrity is maintained – keeping the potentially harmful biochemical elements, which the cell is harboring, from leaking into the extracellular environment and negatively affecting healthy bystander cells, causing inflammation, etc. Hence, the organism is able to recycle the materials from the now defunct cell for future generations of cells. In contrast, oncosis, the process through which a cell undergoes necrotic cell death, is characterized by karyolysis and cell swelling. Necrotic cell
death leads to an explosion of intracellular matter into the extracellular environment. Hence, necrotic cell death can lead to negative effects, like inflammation.

The idea that apoptosis is “programmed” into a cell yields the possibility, by analogy, that we may have the means to reprogram a cell to live or die as needed. In other words, if a cell has cancer, then we can reprogram it to die. If a cell has a virus – e.g., the human immunodeficiency virus or the human papillomavirus – then we can reprogram it to die. In contrast, we may reprogram cells in very close proximity to HIV-infected cells – the so-called bystander cells – to live. These examples may be over-simplified, but they illustrate the importance of understanding genetic manipulation on an in silico effort. If you change one gene, you need to be able to predict the consequences of that change. That is at the heart of computational biology. However, we are not ready to begin manipulating genes involved in apoptosis, because the signaling cascades are too complex. There are too many interacting elements with multiple responsibilities. The only way to gain a deep understanding of the molecular interactions involved in apoptotic signaling is through the development of extensive computer models.

4.1 Apoptotic Signaling Cascades

The study of apoptotic signaling cascades is especially interesting to the fields of biology and medicine since defects in these pathways have been linked to various autoimmune disorders [72], neurological disorders [84], and cancers [42, 82]. Indeed, a recent study [130] implicates Fas-mediated apoptosis in patients with spinal cord injury. Therefore, understanding the molecular mechanisms underlying the apoptosis
pathways can offer new therapeutic approaches to combating a wide range of diseases and disorders.

There are two signaling pathways for apoptosis described in the literature: the extrinsic and the intrinsic (or mitochondrial) pathway. The physiological response induced by each pathway is the same – DNA fragmentation, degradation of cytoskeletal and nuclear proteins, formation of apoptotic bodies, etc. However, albeit the two pathways have distinct beginnings, the molecular mechanisms in the final steps of the signaling cascades are the same. The two pathways converge with particular members of a family of cysteinyl-aspartate-specific proteases, caspases, which are produced as zymogens – i.e., they require a biochemical change to become active [69].

The intrinsic apoptotic pathway is initiated in response to some type of intracellular event – e.g., DNA damage from radiation. Typically, this pathway leads to the induction of the p53 pathway – a DNA repair pathway. If the DNA cannot be sufficiently repaired, a signaling cascade involving p53 leads to reduced mitochondrial permeability, formation of the apoptosome complex, and activation of the effector caspases. The activation of the effector caspases – for example, Caspase 3 – are important for the morphological changes associated with apoptosis [95].

For the extrinsic pathway, cells can receive apoptotic signaling via autocrine and paracrine messaging. That is, the cell can receive a death signal from a neighboring cell, or it can send itself a death signal. These death signals come in the form of apoptotic stimuli which bind to transmembrane receptors – the so-called death receptors – on the cell surface. The known ligand/receptor combinations are TNF-\(\alpha\)/TNFR1, FasL/FasR, Apo3L/DR3, Apo2L/DR4, Apo2/DR5 [26]. Depending on
the biochemical composition of the cell, these death-inducing stimuli can lead to a proteolytic cascade, whereby the inactive proenzyme caspase are activated and cell death can occur. For instance, Cytotoxic T lymphocytes can send death-inducing ligands (specifically, the Fas ligand) to cells as a method for fighting disease or viral infection. This is the main method through which the body fights disease.

For the rest of this Chapter, we concern ourselves with the molecular mechanisms underlying the Fas-mediated apoptotic signaling cascade – extrinsic and intrinsic pathways.

4.2 Fas-mediated Apoptosis

In the past decade, there has been a wealth of information discovered on the Fas-mediated apoptotic pathway. For instance, in [56] they were able to show that Fas/FasL interactions are required for apoptosis of activated T-cells.

One of the troublesome characteristics of some cancerous cells is the upregulation of Fas ligand. This so-called counterattack, is a method the tumor cells can use to delete (by apoptosis) antitumor lymphocytes [96]. There are a variety of tumor types – e.g., colon cancer, esophageal cancer, melanoma, astrocytoma – showing high expression levels of Fas ligand [97].

In order to test the effectiveness of our technique, we decided to simulate the Fas-mediated signaling cascade, as it was reported in [47]. A graphical representation of the model can be found in Figure 4.1. The rules are found in Appendix D. Next, we will walk you through the Fas-induced apoptotic signaling cascade. The Fas pathway is most accurately described as two different pathways [107] – type I and type II –
sharing an initial phase and an ending phase but unique in the molecular mechanisms in between.

Figure 4.1 Picture of the Fas-mediated apoptotic signaling cascade. Both the type I and type II pathways are illustrated.

Both the type I and type II pathways begin the same way: the Fas ligand (FasL) binds to a transmembrane receptor, Fas (CD95/APO-1). This receptor is a member of the tumor necrosis factor-receptor super family – a family consisting of over 30 proteins interacting with 19 different ligands. It is expressed on a variety of cells including activated T and B cells. The binding of ligand to receptor is known as receptor cross-linking. When this cross-linking occurs, a conformational change takes place in the receptor producing the complex Fasc. The crosslinking between ligand and receptor, along with recruitment of Fas-associated death domain, form the components of the Death-Inducing Signaling Complex (DISC) [62]. The cytoplasmic
domain of this complex recruits Fas-associated death domain (FADD), with a maximum of three FADD per binding site (trimer). While FADD is bound to the complex Fasc, Caspase-8 and FLIP are recruited competitively. Once at least two molecules of Caspase-8 have been recruited to a binding site, a dimer, Caspas-8^2, is released into the cytoplasm, where it can then be phosphorylated into active form (Caspase-8*). The binding of FLIP to the Fasc complex is considered to be an inhibitor of apoptosis, because it decreases the number of sites available for Caspase-8 recruitment.

Unless sufficiently inhibited, the signaling cascade can continue in two different ways – the type I or type II pathway. If the initial concentration of Caspase-8 is large enough, Caspase-3 will be directly phosphorylated by the Caspase-8* (the type I pathway). Otherwise, Caspase-8* can truncate molecules of Bid (tBid). Each tBid molecule binds with two Bax molecules, which leads to the release of Cytochrome c from the mitochondria (type II pathway). Once released, Cytochrome c binds to Apaf and ATP, forming a complex that can recruit and phosphorylate Caspase-9 (Caspase-9*). The active Caspase-9* molecules can continue the cascade by direct phosphorylation of Caspase-3. We consider the activation of Caspase-3 to be the end of the signaling cascade. Hence, from our perspective, the cell is dead once all of the Caspase-3 molecules are activated.

Besides FLIP, there are other inhibiting factors at play: Bcl-2 hinders the release of Cytochrome c from the mitochondria and XIAP blocks Caspase-9* from binding with Caspase-3. In other words, if sufficient levels of FLIP, Bcl-2, and/or XIAP exist, the apoptotic pathway can be blocked, and the cell lives.
4.3 Results of Discrete Method

We modeled the pathway described above using 101 different rules working on 53 distinct proteins and protein complexes. Fei Hua et al., in [47], provided the results for the system of ordinary differential equations, as well as some experimental data (from the Jurkat cell line) which they used to fit their model. We compared our results with the results from [47], simulating the same 101 rules and same initial conditions as the system of ordinary differential equations.

Figure 4.2 The decline of full length Caspase-3 for different concentrations of Bcl-2: (i) baseline, (ii) 10-fold increase, and (iii) 100-fold increase. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Similar to [47], we simulated three different initial concentrations for Bcl-2: the baseline value (75nMs), an increase by 10-fold (750nMs), and an increase by 100-fold (7500nMs). Assuming a cell volume of $10^{-12}$ liters, we converted the concentrations into molecular multiplicities: baseline value (45166 molecules), 10-fold (451660 molecules), and 100-fold (4516606 molecules). We expected to see a decline in Caspase-3 activation as Bcl-2 concentration was increased by 10-fold and 100-fold; we provide the results of our simulations in (Figure 4.2). We also provide the results
of simulations with a decrease of 10-fold and 100-fold in comparison to the baseline Bcl-2 multiplicity (Figure 4.3). Notice, the graph based on the NWT algorithm is comparable to the ODE-based results from [47].

![Graphs showing the decline of full length Caspase-3 for decreased concentrations of Bcl-2: (i) 10-fold and (iii) 100-fold. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.]

Figure 4.3 The decline of full length Caspase-3 for decreased concentrations of Bcl-2: (i) 10-fold and (iii) 100-fold. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

### 4.4 Bcl-2’s Effects on the Type II Pathway

Next, we analyzed the Caspase-3 activation kinetics by considering the different mechanisms through which it has been suggested that Bcl-2 blocks the type II pathway. In [16], [87], and [123] the authors suggested that Bcl-2 might bind with (a) Bax, (b) Bid, (c) tBid, or (d) both Bax and tBid to block the mitochondrial pathway. We implemented four different sets of rules to test each Bcl-2 binding mechanisms. We refer the interested reader Appendix D for the details of the rules.

The dynamics of Caspase-3 activation were studied by increasing the baseline Bcl-2 concentration by 10-fold and 100-fold. The conclusion of [47] is that Bcl-2 binding to both Bax and tBid (d) is the most efficient mechanism for inhibiting apoptosis.
Our Membrane System agrees with the observations from [47]. The results of (d) are illustrated in Figure 4.2, and (a) - (c) can be seen in Figure 4.4 - Figure 4.6. A comparison of (a) - (d) at baseline Bcl-2 concentration is shown in Figure 4.7.

![Graph showing the effects of Bcl-2 binding to Bax only.](image)

**Figure 4.4** The effects of Bcl-2 binding to Bax only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

![Graph showing the effects of Bcl-2 binding to tBid only.](image)

**Figure 4.5** The effects of Bcl-2 binding to tBid only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

### 4.5 Modeling the Behavior of the Type I Pathway

**Algorithm**

Some cells are not sensitive to Bcl-2 over expression, as described in [107]. In these cells, Caspase-3 is activated through the type I pathway, bypassing the role of
the mitochondria and Bcl-2. Scaffidi et al. have suggested in [107] that the type of pathway is chosen based on the concentration of Caspase-8 generated in active form following the binding of Fas ligand to its receptor site. High concentration of active Caspase-8 allows for direct activation of Caspase-3 (type I), but if the concentration of Caspase-8 is sufficiently low, amplification of the death signal through the mitochondria is required to induce cell death (type II). We tested this hypothesis by increasing the initial concentration of Caspase-8 by 20-fold (from 33.33nMs to 666.6nMs), which was expected to lead to increased active Caspase-8* throughout the simulation run.

![Graphs showing the effects of Bcl-2 binding to Bid only.](image)

Figure 4.6 The effects of Bcl-2 binding to Bid only. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

We ran two different versions of the increased Caspase-8 model, using the baseline concentration of Bcl-2 and an increase of Bcl-2 by 100-fold, in order to gauge the sensitivity of the type I pathway to Bcl-2 upregulation. Figure 4.8 shows that Caspase-3 activation was not sensitive to the increase in Bcl-2 concentration, which is the hallmark for type I pathway dominant behavior. N.B., for these simulations Bcl-2 was allowed to bind to both Bax and tBid, which was shown above to be the most efficient mechanism for Bcl-2 inhibition of apoptosis.
Figure 4.7 The results of baseline Bcl-2 concentration with each of the four mechanisms for Bcl-2 inhibition (binding with Bax only, Bid only, tBid only, or Bax and tBid). (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Figure 4.8 Investigation of the effects of Bcl-2 increase (100-fold) for the type I pathway. (a) The results of the solution to the system of ordinary differential equations and (b) the results of the NWT algorithm.

Our Membrane System has yielded results comparable to the solutions to the system of ordinary differential equations. The sixteen distinct simulations show similar apoptotic behavior to the deterministic results (Figure 4.2 through Figure 4.8). However, albeit the activation of Caspase-3 is similar between the two techniques, the molecular interactions throughout are different. We have compared the results of our simulator with the experimental results in [47], the deterministic results from the
same paper, and the stochastic approach described in [17]. The Caspase-3 results are as expected, but the activation of Caspase-8 raises our interest. See Figure 4.9 for a comparison between the techniques.

Figure 4.9 Results of the three simulation techniques and experimental data provided by [47], showing decline of (a) full length Caspase-3 and (b) full length Caspase-8.
The experimental data and deterministic results were obtained from Fei Hua et al. We see that the decrease of full length Caspase-3 is similar in all three simulation results. Interestingly, the decline of full length Caspase-8 is less prominent in the two Membrane System simulations. The contrast could be the result of the discrete nature of the Membrane Systems. As for both results being different than the experimental data, we believe that further investigation of kinetic rates of the reactions will allow for better agreement between simulation and experimentation.

4.6 Summary

We have chosen to simulate Fas-induced apoptosis because it has one of the most detailed descriptions/characterization in the literature (due in large part to its role in cancer and HIV research). In the interest of comparing our Membrane System with the solutions to the system of ordinary differential equations, we have implemented 101 different rules working on 53 distinct proteins and protein complexes. The pathway begins with the stimulation of FASL and ends with the activation of the effector Caspase-3. Fei Hua et al., in [47], provide the deterministic results, as well as some experimental data (from the Jurkat cell line), which they used to fit their model.

The consistency between the framework and the experimental results of [47] validates our model. Our NWT algorithm shows that Membrane Systems are an intriguing alternative to ordinary differential equations methods. We have argued that the discrete nature of our technique might be better for simulating the evolution of systems involving low numbers of molecules.

In Chapter 5, we will build on the rules for Fas-mediated apoptosis discussed in
this Chapter. There are a few proteins encoded in the HIV genome, which seem to have severe consequences on Fas-induced apoptosis [109]. The so-called 'latently' infected T cells are especially interesting in the potential strategies for the eradication of the AIDS epidemic [41].
CHAPTER 5

HIV-1 EFFECTS ON THE FAS PATHWAY

In Chapter 5 we will explore the qualities of the human immunodeficiency virus (HIV) which help it remain remarkably difficult to cure. Our goal is to model the effects of HIV-1 proteins on the Fas-induced apoptotic pathway. This is the first effort of its kind, and we hope it will provide insight into future HIV research and modeling.

The virus has several remarkable qualities: (1) it predominantly infects the cells of the immune system; (2) it shows a high genetic variation throughout the infection in a single individual due to the high error rate in the reverse transcription; (3) it induces apoptosis in the so-called bystander immune cells; and (4) normal immune system function can cause some HIV-infected T cells to become latent, entering a reversibly nonproductive state of infection. Since the latent cells are transcriptionally silent, they are virtually indistinguishable from the uninfected cells. Also, the number of latently infected cells is relatively small, which makes the experimental study of these cells difficult – current technology in biochemistry requires large numbers of the molecules/cells to be studied. It is widely believed that the latently infected CD4+ T cells represent the last barrier to an HIV cure. This Chapter is based on our publication in WMC09, presenting a first modeling effort for the Fas-mediated
apoptosis (or programmed cell death) of latently infected T cells [51].

We will focus on the apoptotic modeling (reason 3), since it is the avenue through which the virus destroys the effectiveness of the host's immune system. We will base our model on the work described in Chapter 4, using the Nondeterministic Waiting Time algorithm discussed in Chapter 2. Furthermore, in order to make the modeling effort easier and due to the high genetic variability (reason 2) of the viral genome, we will combine several similar processes together into single reactions. The kinetic constants for the new reactions, modeling the biochemical interactions involving viral proteins with the host cell, will be obtained by fitting the model to reported experiments on the infected, nonlatent cells. Finally, we will simulate the reactivation of latently infect T cells by making some adjustments to the appropriate initial conditions of the system.

5.1 A Brief History of HIV

HIV, which is responsible for the onset of acquired immune deficiency syndrome (AIDS), has lead to more deaths than nearly any other virus in human history. Indeed, AIDS is called a global pandemic by the World Health Organization (WHO), and is "undoubtedly the defining public-health crisis of our time" [111]. According to statistics from the WHO, there were 33.2 million people living with HIV in 2007, 2.5 million newly infected individuals, and 2.1 million AIDS deaths [126].

When AIDS was first labeled as a diseases in 1981, there was an initial debate over what was causing the immune syndrome. It was suggested that a retrovirus could be the cause. However, retroviruses were a relatively new field of study and only a
few had yet been identified. Two labs, Gallo’s in the United States and Montagnier’s in France, are both credited with the discovery of HIV [7, 78]. Although political tensions surrounded the discovery, the two groups essentially agree that it was a combined effort to discover the virus behind AIDS.

Upon discovery of the virus causing AIDS, there was an explosion of subsequent breakthroughs. A bloodtest was developed within a couple of years [77], securing the world’s blood transfusion supply. In 1985, a group successfully sequenced the viral genome [122]. And, the first anti-HIV drug, AZT, was developed by 1987 [77]. With such quick discoveries, some believed the race for a cure would be a short one.

However, this task quickly became quite daunting, for reasons we will elaborate in the rest of Section 5.1, and the race for the cure has turned into an everlasting marathon. Even though there are a variety of drugs now available to combat the effects of HIV, reducing it to a sort of chronic illness, the complete eradication of the viral infection is still yet an unattained achievement for science. There are two important qualities of HIV which makes it difficult to find a cure: (a) a high genetic variability and (b) an ability to go silent (so-called HIV latency).

One of the reasons the cure for HIV remains elusive is the high genetic variability of the virus. There are two strains of HIV: type 1 and type 2. HIV-1 is the virus whose discovery was discussed above. HIV-2 was specifically isolated [21] in West Africa in 1986. Unlike HIV-1, the type 2 strain remains confined to West Africa [106]. Since HIV-1 is more virulent and transmissive [106], the discussion and modeling efforts of this Chapter will be concerned with the type 1 variant.

HIV-1 can be broken down into multiple subtypes (see Figure 5.1). Infected indi-
viduals are susceptible to co-infections and superinfections [111]; this can lead to new recombinant forms of the virus. For instance, in Southeast Asia an estimated twenty percent of infections come from recombinant forms [111]. Lack of immunization and continuous evolution of the viral genome makes vaccine development a considerable challenge. The modeling community must remain aware of the different subtypes, to avoid illfit models based on these genetically distinct subtypes, resulting in poor approximation of reality.

Figure 5.1 Estimated number of HIV-1-infected individuals in 2007 [126] as well as the various HIV-1 subtypes per region [111].

5.2 AIDS Pathogenesis

The pathogenesis of AIDS is attributed to the depletion of the host’s CD4+ T cells, the loss of which results in a dysfunctional immune system. Finkel and colleagues in [28] concluded that HIV-1 infection causes death predominantly in the so-called bystander T cells. These healthy, uninfected cells are marked for destruction by the neighboring HIV-1-infected cells. The mechanism of the bystander cell death was
shown to be apoptosis. Proteins encoded by the HIV-1 genome exhibit anti- and pro-apoptotic behavior on infected and bystander cells, enhancing or inhibiting a cell’s ability to undergo apoptosis.

There are numerous drugs available for limiting the impact of HIV-1 on the immune system; the most successful approach, highly active anti-retroviral therapy (HAART), is a combination of several types of drugs, targeting different mechanisms of HIV-1 infection and proliferation. Although HAART has proven to be effective in the reduction or elimination of viremia [89], it is ineffective in the complete eradication of the viral infection. The HIV-1 infection is able to persist in a dormant state throughout the entire time a patience is on HAART. The way this is accomplished is one of the most remarkable qualities of HIV – i.e., latency.

Latent reservoirs of HIV-1 have been detected in HIV-1-infected patients [18, 19]. Latently infected cells are relatively rare – about 1 in $10^6$ resting T cells [19]. However, they are considered to be the largest obstacle in combating HIV-1 infection [29, 110, 114]. Understanding the mechanisms behind HIV-1 latency is a focal point for current AIDS-related research (for a recent review on latency see [41]).

There are two types of latency described in the literature. The first, preintegration latency, refers to resting T cells containing unintegrated HIV-1 DNA. If a T cell is in a resting state, the HIV-1 DNA is not able to quickly integrate into the host’s viral genome. Since the unintegrated HIV-1 DNA is labile and reverse transcription of HIV-1 RNA is slow (on the order of days) [90, 131, 132, 135], it is believed that patients with reduced viremia after several months of HAART therapy do not have resting T cells with unintegrated HIV-1 DNA [11]. Hence, we will not concern ourselves with
modeling preintegration latency. We will instead discuss the second form of latency — postintegration latency. It is this alternative form of latency which will be the focus of the rest of the Chapter.

Postintegration latency refers to resting T cells with stably integrated HIV-1 DNA. These cells can provide a reservoir for viral reproduction for years [29]. The cells exist as a natural consequence to normal healthy immune function. When the body is invaded by an organism, T cells are activated to destroy the invading pathogen. Once the pathogen is destroyed, many of the T cells commit apoptosis or else they would persist in killing other cells at the inevitable detriment of the host organism. However, a few of these active T cells return to a quiescent state. This return to a nonactive status is the basis for so-called memory T cells.

When an activated HIV-1-infected T cell turns into a memory T cell, this is very troublesome for the infected individual. The individual now has a T cell which is virtually indistinguishable from all of the other resting T cells, and yet it is infected with the HIV-1 genome. The cell can persist almost indefinitely in this state. Hence, when the individual goes off HAART and the resting HIV-1-infected T cell is reactivated, viraemia is quickly restored and the individual will succumb to AIDS.

It is because of their long lifespan and ability to restore viraemia that we have chosen to model the reactivation of a postintegration latently infected CD4+ T cell. We chose to model the Fas-induced apoptosis of these cells because the effects are well characterized in the literature and no one has made an attempt to do so before.

As far as we know, this paper reports the first attempt at modeling the Fas-mediated apoptotic signaling pathway in reactivated latently infected CD4+ T cells.
We will draw upon the system we laid out in Chapter 4, which is based on information for the Jurkat T cell line from [47] (and references therein). We have extended the model from Chapter 4 in order to better understand the reactivation of latently infected T cells.

5.3 HIV-1 Infection

There is still some debate about the effects of HIV-1 proteins on cellular signaling networks; however, we have pooled the collective knowledge of the biological community in order to categorize and model the described functions of various HIV proteins. For an illustration of the Fas pathway and the involvement of the HIV proteins we refer the reader to Figure 5.2. We will explain the inspiration for the model below.

The mechanisms behind HIV-1 infection of CD4+ T cells are well understood. A spike on the virus, the gp120 envelope glycoprotein, binds to the CD4 receptor of the target cell and, in conjunction with subsequent binding to a coreceptor (CCR5 or CXCR4), a path is opened for the virus to inject its contents into the cell [15, 128]. Reverse transcriptase creates cDNA from the HIV-1 RNA and the genome of the virus is implanted into the cell’s own DNA for future production. During this time, the immune system fails to detect and destroy the infected cell.

Upon infection, the contents of the virion (e.g., Vpr, HIV protease (HIV_{pr}), reverse transcriptase (RT), and HIV RNA (HIV_{RNA})) are released into the cytoplasm [14]. In the newly infected and active CD4+ T cells, the HIV_{RNA} is converted to cDNA (HIV_{cDNA}) by the reverse transcriptase about five hours post-infection [60]. The HIV_{cDNA} is then integrated into the host’s genome with the help of the viral integrase.
approximately one hour later [27]. We will formalize these rules as we discuss the inspiration from the literature behind their creation. For our convenience, we have labeled the integrated HIV genome as $HIV_{LTR}$ in our rules. $HIV_{LTR}$ is the basis for interactions involving the HIV long terminal repeat; in our model, it is a necessary component for all reactions pertaining to HIV-1 protein production.

Figure 5.2 HIV-1 protein effects on the Fas-mediated signaling cascade.

After integration of the viral DNA, gene expression of HIV proteins becomes possible. The nuclear factor of activated T cells (NFAT) and NF-κB have been shown
to play important roles in HIV gene expression [61, 80]. In a resting CD4+ T cell, NF-κB is sequestered in the cytoplasm by its inhibitor, IκB. Following cellular activation, NF-κB is released by its inhibitor, which allows it to relocate to the nucleus where it can bind to the HIV_LTR. Also following T cell activation, NFAT, located in the cytoplasm of resting CD4+ T cells, undergoes dephosphorylation and translocation to the nucleus where it can bind to the HIV_LTR [61]. Once NF-κB and NFAT are translocated to the nucleus, they can bind to the HIV_LTR, combining their efforts to synergistically enhance the promoter activity. Moreover, [61] shows that the combined effects of Tat, NF-κB and NFAT is much stronger than the pairings of Tat and NF-κB or Tat and NFAT. In our model, we have combined the roles of NF-κB and NFAT. The translocation and binding rules for NFAT (and NF-κB) are shown in Table 5.2.

Table 5.1 Reactions involving translocation of the HIV genome and integration into the host’s genome.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: HIVRNA + RT → HIV_cDNA + RT</td>
<td>$k_{21}$</td>
</tr>
<tr>
<td>2: HIV_cDNA → HIV_cDNA (nuclear import)</td>
<td>$k_{22}$</td>
</tr>
<tr>
<td>3: HIV_cDNA → HIV_LTR</td>
<td>$k_{22}$</td>
</tr>
</tbody>
</table>

Multiply spliced (MS) HIV-1 mRNAs – responsible for Tat/Rev protein creation – are detectable in resting CD4+ T cells [68]. However, due to the inefficient export of the mRNA transcripts to the cytosol, Tat and Rev proteins are undetectable in the latent cells. Activation of these latent cells leads to production of Tat and Rev, and subsequent upregulation of all HIV-1 proteins. In order for the infected cells to create HIV proteins other than Tat and Rev, the transcriptional elongation induced by Tat and the efficient nuclear export of MS HIV-1 RNAs by Rev are required. Our latent
cell model, beginning with cellular activation, initially allows for inefficient creation of Tat proteins. We chose not to model Rev, since it has no known Fas apoptotic function; its exporting functions are incorporated into the kinetic constants governing mRNA translocation. Once Tat is located in the nucleus, it requires the help of two other proteins provided by the host cell: CyclinT1 and CDK9.

Table 5.2 Reactions involving HIV long terminal repeat (LTR) and HIV mRNA production.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4: NFAT → NFAT (nuclear import)</td>
<td>( k_{23} )</td>
</tr>
<tr>
<td>5: CDK9 → CDK9 (nuclear import)</td>
<td>( k_{24} )</td>
</tr>
<tr>
<td>6: CyclinT1 + CDK9 → PTEFb</td>
<td>( k_{25} )</td>
</tr>
<tr>
<td>7: NFAT + HIV(<em>{LTR}) → HIV(</em>{LTR}):NFAT</td>
<td>( k_{26} )</td>
</tr>
<tr>
<td>8: HIV(<em>{LTR}):NFAT + Tat → HIV(</em>{LTR}):NFAT:Tat</td>
<td>( k_{27} )</td>
</tr>
<tr>
<td>9: HIV(<em>{LTR}):NFAT:Tat + PTEFb → HIV(</em>{LTR}):NFAT:Tat:PTEFb</td>
<td>( k_{28} )</td>
</tr>
<tr>
<td>10: HIV(<em>{LTR}) → HIV(</em>{LTR}) + mRNA(_{Tat}^)</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>11: HIV(<em>{LTR}) → HIV(</em>{LTR}) + mRNA(_{Vpr}^)</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>12: HIV(<em>{LTR}) → HIV(</em>{LTR}) + mRNA(_{Nef}^)</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>13: HIV(<em>{LTR}) → HIV(</em>{LTR}) + mRNA(_{Tat}^)</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>14: HIV(<em>{LTR}):NFAT → HIV(</em>{LTR}):NFAT + mRNA(_{Vpr}^)</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>15: HIV(<em>{LTR}):NFAT → HIV(</em>{LTR}):NFAT + mRNA(_{Vpr}^)</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>16: HIV(<em>{LTR}):NFAT → HIV(</em>{LTR}):NFAT + mRNA(_{Vpr}^)</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>17: HIV(<em>{LTR}):NFAT → HIV(</em>{LTR}):NFAT + mRNA(_{Nef}^)</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>18: HIV(<em>{LTR}):NFAT:Tat → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Tat}^)</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>19: HIV(<em>{LTR}):NFAT:Tat → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Vpr}^)</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>20: HIV(<em>{LTR}):NFAT:Tat → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Vpr}^)</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>21: HIV(<em>{LTR}):NFAT:Tat → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Nef}^)</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>22: HIV(<em>{LTR}):NFAT:Tat + PTEFb → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Tat}^)</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>23: HIV(<em>{LTR}):NFAT:Tat + PTEFb → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Vpr}^)</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>24: HIV(<em>{LTR}):NFAT:Tat + PTEFb → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Vpr}^)</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>25: HIV(<em>{LTR}):NFAT:Tat + PTEFb → HIV(</em>{LTR}):NFAT:Tat + mRNA(_{Nef}^)</td>
<td>( k_{32} )</td>
</tr>
</tbody>
</table>
In an inactivated cell, CyclinT1 and CDK9 are sequestered in the cytoplasm [81]. Upon T cell activation, they are relocated to the nucleus. CyclinT1 and CDK9 combine to make up the positive-acting transcription elongation factor (P-TEFb) complex. The binding of P-TEFb and Tat at the HIV\textsubscript{LTR} allows the hyperphosphorylation of RNA polymerase II (RNAPII), resulting in increased transcriptional elongation. The translocation and binding rules for CyclinT1, CDK9 and Tat are formalized in Table 5.2. The transcription, translocation, and translation rules involving HIV-1 mRNA molecules are also summarized in Table 5.3.

Table 5.3 Translation and degradation rules for HIV mRNA.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>26: mRNA\textsubscript{Tat} → mRNA\textsubscript{Tat} (nuclear export)</td>
<td>$k_{33}$</td>
</tr>
<tr>
<td>27: mRNA\textsubscript{Nef} → mRNA\textsubscript{Nef} (nuclear export)</td>
<td>$k_{33}$</td>
</tr>
<tr>
<td>28: mRNA\textsubscript{Vpr} → mRNA\textsubscript{Vpr} (nuclear export)</td>
<td>$k_{33}$</td>
</tr>
<tr>
<td>29: mRNA\textsubscript{HIV\textsubscript{Vpr}} → mRNA\textsubscript{HIV\textsubscript{Vpr}} (nuclear export)</td>
<td>$k_{33}$</td>
</tr>
<tr>
<td>30: mRNA\textsubscript{Tat} → mRNA\textsubscript{Tat} + Tat</td>
<td>$k_{34}$</td>
</tr>
<tr>
<td>31: mRNA\textsubscript{Nef} → mRNA\textsubscript{Nef} + Nef</td>
<td>$k_{34}$</td>
</tr>
<tr>
<td>32: mRNA\textsubscript{Vpr} → mRNA\textsubscript{Vpr} + Vpr</td>
<td>$k_{34}$</td>
</tr>
<tr>
<td>33: mRNA\textsubscript{HIV\textsubscript{Vpr}} → mRNA\textsubscript{HIV\textsubscript{Vpr}} + HIV\textsubscript{pr}</td>
<td>$k_{34}$</td>
</tr>
<tr>
<td>34: mRNA\textsubscript{Tat} → degraded</td>
<td>$k_{35}$</td>
</tr>
<tr>
<td>35: mRNA\textsubscript{Nef} → degraded</td>
<td>$k_{35}$</td>
</tr>
<tr>
<td>36: mRNA\textsubscript{Vpr} → degraded</td>
<td>$k_{35}$</td>
</tr>
<tr>
<td>37: mRNA\textsubscript{HIV\textsubscript{Vpr}} → degraded</td>
<td>$k_{35}$</td>
</tr>
</tbody>
</table>

5.4 HIV-1-Related Effects on the Fas Pathway

Aside from its role in transcriptional elongation, the Tat protein is responsible for both pro- and anti-apoptotic behavior. In [8], the authors demonstrated that increased Tat expression causes upregulation of inactive Caspase-8, which is a pro-apoptotic molecule. Also, Tat has been associated with the downregulation of Bcl-2
[109], which is an anti-apoptotic molecule. Given the pro- and anti-apoptotic duties of Caspase-8 and Bcl-2, respectively, it would appear that a cell with high levels of Tat has increased susceptibility to apoptosis. Conversely, [28] claims that Tat upregulates Bcl-2, resulting in decreased apoptotic rates of cells. Tat has also been implicated in the upregulation of Fas ligand on the cell surface [8, 129], which may effect the cell through autocrine signaling. The anti- and pro-apoptotic rules for Tat are found in Table 5.4.

Table 5.4 Reactions involving Tat protein.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>38: Tat ⇌ Tat (nuclear import/export)</td>
<td>( k_{36f}, k_{35r} )</td>
</tr>
<tr>
<td>39: Tat → Tat + Casp8</td>
<td>( k_{37} )</td>
</tr>
<tr>
<td>40: Tat → Tat + Bcl2</td>
<td>( k_{38} )</td>
</tr>
<tr>
<td>41: Tat → FasL + Tat</td>
<td>( k_{39} )</td>
</tr>
<tr>
<td>42: Tat + Bcl2 → Tat</td>
<td>( k_{40} )</td>
</tr>
</tbody>
</table>

The HIV-1 Vpr has been shown to both enhance and inhibit the Fas signaling cascade. Upon infection, the ~700 molecules of Vpr in the virion are injected into the cytoplasm of the cell [14]. At low levels, Vpr has been shown to prohibit apoptosis by upregulating Bcl-2 and downregulating Bax [22]. However, higher concentrations of Vpr affect the mitochondrial membrane permeability via interactions with the permeability transition pore complex (PTPC), resulting in the release of Cytochrome c into the cytoplasm [53]. In the same paper, the authors also demonstrated that Bcl-2 can inhibit the effects of Vpr on the PTPC. The various apoptotic roles of Vpr we define in Table 5.5.

Another protein packaged in HIV-1 virions, HIVpr, plays an important role in the
Fas pathway. The HIV$_{pr}$ has been shown to cleave Bcl-2 into a deactivated state [113], while it also cleaves Caspase-8 [83] into active form. Both rules are pro-apoptotic and are in Table 5.6.

Table 5.5 Reactions involving Vpr protein.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>43: Vpr + Bax $\rightarrow$ Vpr</td>
<td>$k_{43}$</td>
</tr>
<tr>
<td>44: Vpr + Bcl2 $\rightarrow$ Vpr:Bcl2</td>
<td>$k_{44}$</td>
</tr>
<tr>
<td>45: Vpr:Bcl2 $\rightarrow$ Vpr + Bcl2</td>
<td>$k_{45}$</td>
</tr>
<tr>
<td>46: Vpr + PTPC $\rightarrow$ Vpr:PTPC</td>
<td>$k_{46}$</td>
</tr>
<tr>
<td>47: Vpr:PTPC + Cyto.c $\rightarrow$ Cyto.c* + Vpr:PTPC</td>
<td>$k_{47}$</td>
</tr>
</tbody>
</table>

Table 5.6 Reactions involving HIV protease.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>48: HIV$<em>{pr}$ + Casp8 $\rightarrow$ HIV$</em>{pr}$ + Casp8*</td>
<td>$k_{48}$</td>
</tr>
<tr>
<td>49: HIV$<em>{pr}$ + Bcl2 $\rightarrow$ HIV$</em>{pr}$</td>
<td>$k_{49}$</td>
</tr>
</tbody>
</table>

Finally, we define two pro-apoptotic rules for the Nef protein. Zauli et al. discovered in [134] that Nef can play a role in cell death by upregulating Fas receptor and Fas ligand on the cell surface. Upregulating the receptor sites of Fas on the cell surface prepares the cell for ligand binding, and can initiate the Fas-induced apoptotic signaling cascade. The upregulation of Fas ligand may protect the infected cell from cytotoxic T cells, or it could be part of autocrine signaling. The four rules for upregulation and translocation of Fas and Fas ligand are in Table 5.7.

5.5 Modeling Results

We added all of the rules from Tables 5.1-5.7 to the Fas model described in Chapter 4 – see Appendix D for the complete list. From this, we are able to simulate two types
of cells: nonlatent and latent. The differences between the two models are the initial protein multiplicities. The nonlatent cell is an activated T cell which has just been infected with the contents of the HIV-1 virion. The HIV-1 RNA and other viral proteins are in the cytoplasm. The HIV-1 RNA must be incorporated into the host's genome before the viral protein production can begin. The latent model is a newly activated T cell with no HIV-1 proteins present. However, the HIV-1 genome is already integrated into the host's DNA.

Table 5.7 Reactions involving Nef protein.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Reaction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50: Nef $\rightarrow$ Nef + Fas</td>
<td>$k_{49}$</td>
</tr>
<tr>
<td>51: Nef $\rightarrow$ Nef + FasL</td>
<td>$k_{50}$</td>
</tr>
<tr>
<td>52: FasL $\rightarrow$ FasL (to cell surface)</td>
<td>$k_{51}$</td>
</tr>
</tbody>
</table>

As we have discussed earlier, the nonlatent cell is used for the model fitting, since the majority of information about HIV-1 proteins pertains to these types of cells. The reason for this lies in the fact that latent cells are transcriptionally silent (virtually undetectable) and relatively rare. For instance, in Figure 5.3(a), the results from the nonlatent simulation show the activity of Tat in that full length (inactive) Caspase-8 increases by a factor of three. Our simulation agrees with the observations of [8]. Also, in Figure 5.3(b), our model shows Vpr-induced upregulation of Bcl-2 and downregulation of Bax by 30% and 20%, resp. Our results agree with the experimental results on Vpr described in [22].

We will next consider the activation of Caspase-3. In Figure 5.4, both the nonlatent and latent models are shown to exhibit the onset of apoptosis – total activation
of Caspase-3 — after approximately two days. Our results indicate that reactivated latently infected CD4+ T cells activate all of the Caspase-3 molecules earlier than the nonlatent model. Also, in Figure 5.4, we show the truncation of Bid, which is a necessary step in the induction of the type II pathway. Active Caspase-8 is responsible for the truncation of Bid, so we are seeing the downstream effects of Caspase-8 activation.

![Graphs showing the upregulation of Caspase-8 and Bcl-2 and downregulation of Bax by Vpr](image)

Figure 5.3 (a) Tat protein upregulates Caspase-8 levels by three-fold. (b) Vpr upregulates Bcl-2 and downregulates Bax by 30% and 20%, respectively.

Next, let us consider the mechanisms behind Caspase-3 activation in the latent and nonlatent models. According to the rules in Appendix A, an interaction between full length Caspase-3 and active Caspase-8 or Caspase-9 can have two outcomes: the activation of Caspase-3 or not. Both of our models show cooperation between the two pathways, which is not explicitly stated in the literature. The nonlatent results (Figure 5.5) show the first interactions between Caspase-3 and Caspase-8* molecules occur just after 18 hours into the run. It isn’t until ~ 10 hours later (26 hours into the run) that we begin to see Caspase-3 interactions with Caspase-9*,
after signal amplification through the mitochondria. As discussed in [47, 50], given a sufficiently high initial concentration of Caspase-8 in the cell, signal amplification is not necessary to induce apoptosis. For this model, we set the initial level of Caspase-8 to be insufficient for apoptosis by the type I pathway.

Figure 5.4 (a) Total reduction of full length Caspase-3 is seen after ~ 40 hours in the latent model, whereas the nonlatent model takes ~ 47 hours. (b) The decline of Bid through interactions with Caspase-8, leading to a rise in tBid.

Figure 5.5 Interactions involved in the type I and type II pathways for the nonlatent simulation. (a) The results for the three days of simulation and (b) An excerpt of one minute from the three day simulation (from 32 hours to 32 hours and 1 minute).
The results of the *latent* simulation are similar to the *nonlatent*, where both pathways appear to govern Caspase-3 activation. In the *latent* run (Fig. 5.6), we see type I interactions first occur about 12 hours into the simulation, while type II molecular binding occurs after 21 hours.

![Graph](image1)

**Figure 5.6** Interactions involved in the type I and type II pathways for the *non-latent* simulation. (a) The results for the three days of simulation and (b) An excerpt of one minute from the three day simulation (from 32 hours to 32 hours and 1 minute).

Although Figure 5.5(b) and Figure 5.6(b) imply type I interactions occur more frequently than type II, it must be noted that, due to the kinetics governing these binding rules, Caspase-8* can remain bound to Caspase-3 for a longer period of time than Caspase-9*. Therefore, although it seems that Caspase-8* binds to Caspase-3 more frequently, the reactions are merely slower. In fact, both models exhibit more interactions between Caspase-9* and Caspase-3.

### 5.6 Summary

Based on the biological evidence in the literature, we constructed a model for the effects of HIV-1 proteins on the Fas-mediated apoptosis pathway. This work is the first of its kind, simulating Fas-induced apoptosis in reactivated latently infected
CD4+ T cells. We have provided some preliminary results in an effort to understand CD4+ T cell latency. Interestingly, our results show a cooperation between the type I and type II pathways. We have not been able to verify an explanation for this in the available literature.

Finally, we would like to note that the experimental information on the latent HIV-1-infected CD4+ T cells is scarce, due to the fact that these cells are found in such small numbers in vivo. Therefore, our model relies heavily on applying the knowledge of activated HIV-1-infected CD4+ T cells. We look forward to new experimental results about these enigmatic cells, which we will use to refine the model.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The focus of this entire dissertation has been the modeling of biochemical reaction networks. We introduced various aspects of modeling algorithms and tools from the literature pertaining to signal transduction simulation. The original ideas in this paper pertain to the Nondeterministic Waiting Time (NWT) algorithm and the effects of HIV-1 proteins on the Fas-mediated apoptotic signaling cascade.

We have provided an exhaustive explanation on the design and implementation of the NWT algorithm. We discussed important improvements over the previous modeling efforts by Dr. Păun’s group – specifically, reaction memory, efficient sorting of reaction Waiting Times, and nondeterminism. The algorithm is designed to be faster than the Gillespie Algorithm. And yet, we are able to simulate biochemical systems discretely with a degree of nondeterminism, differentiating our technique from the solutions to systems of ordinary differential equations.

Using our NWT algorithm, we have shown results for modeling some popular networks – e.g., the Lotka-Volterra predator-prey model and a circadian rhythm model. With these two models, we have shown our algorithm can exhibit Gillespie-like results with the latter, while NWT simulations of the former model show similarities to the
results of the solution to the system of ordinary differential equations. However, the computational load of the nondeterministic decisions in our NWT algorithm is far less than the Gillespie Algorithm, as is the case in the noise-induced oscillations for the circadian rhythm model.

We have used our NWT algorithm to explore Fas-induced apoptosis. We started by simulating a model developed by the Lauffenberger lab at M.I.T. [47]. Our simulator is capable of showing comparable results to the solution of the system of ordinary differential equations from that group. However, when compared with the experimental results, we did notice some differences between our discrete, nondeterministic technique, the ordinary differential equations, and experimental results provided by the M.I.T. lab. Activation of Caspase 3 was very close, but Caspase 8 was a bit different. In other words, the end of the signaling cascade occurred at the same time, but a critical earlier component showed different activity. We concluded that this was due to the discrete nature of our simulation.

After an extensive literature review, we were able to extend the Fas model of [47] to incorporate HIV-1 activity. This was the first attempt someone has made in modeling Fas-induced apoptosis in HIV-1-infected cells. We made a special effort to model the so-called latently infected T cells, which are considered the last barrier in the eradication of HIV-1-infection. There are some interesting directions to go with this research.

6.1 Extensions on the HIV Model

There are several avenues to explore in extending the HIV model from Chapter
5. For instance, we would like to model the effects of HIV-1 proteins on bystander cell apoptosis. As mentioned in the introduction, HIV-1 appears to primarily kill uninfected bystander T cells [28]. Various mechanisms have been reported for the destruction of the bystander cells. Along with Fas-induced apoptosis, other possible mechanisms for bystander cell death are reviewed in [3, 105, 109]. Upon being exocytosed by an infected cell, several of the proteins encoded in HIV-1 can exhibit destructive qualities when interacting with neighboring bystander cells – either on the surface or through endocytosis. In Figure 6.1, we see a proposed model for bystander cell apoptosis.

![Figure 6.1](image)

Figure 6.1 The effects of an HIV-1-infected T cell on its neighboring healthy, bystander cells.

There are a few HIV-1 proteins we have ignored in this model, because they affect T cells in ways not within the scope of our current efforts. For example, soluble and
membrane-bound Env can bind to the CD4 receptor of bystander cells. In [20] and [9], the authors have shown that ligation of the CD4 receptor by Env, is sufficient to increase apoptosis in bystander cells. The reasons for the increased apoptotic rates following Env-CD4 binding can be attributed to Bcl-2 down-regulation [44], increased Caspase 8 activation [2], and upregulation of Fas [88], FasL and Bax [109].

Extracellular Tat can enter bystander cells through endocytosis, which leads to pro-apoptotic activity. The addition of Tat to a culture of uninfected cells has been shown to increase apoptosis [74]. Endocytosed Tat can upregulate levels of Caspase 8 [8] and increase expression of the Fas ligand [109], interfering in the same manner as in the infected cells. Also, extracellular Vpr can disrupt the mitochondrial membrane, leading to increased translocation of Cytochrome c* [109].

Modeling a cluster of cells would be a possible extension of this model. Using MPI, we can have each node of a cluster model a distinct cell. One (or more) of the nodes can be an HIV-1-infected cell, while many of the nodes can represent bystander cells. We can simulate the effects of the HIV-infected cell on the healthy, bystander cells. Besides HIV-1-related effects on the Fas-mediated apoptotic pathway, there are other directions to go with Fas modeling.

6.2 Calcium’s Role in Apoptosis

In recent years, calcium’s role in apoptosis has received increased attention. For a recent review, we refer the reader to [93]. It seems that calcium is capable of exhibiting both pro- and anti-apoptotic characteristics. While a large portion of the literature illustrates calcium’s role in the intrinsic (sometimes referred to as the mitochondrial)
pathway, there is also evidence showing its role in the extrinsic pathway.

We propose an exploration into the role of calcium as an apoptotic signaling molecule. The effort will combine experimental and computational techniques to derive new kinetic rates and extrapolate apoptotic signaling behavior for a variety of cell types – besides Jurkat T cells, the lab of Dr. DeCoster is also investigating neurons and astrocytes.

There are several studies showing cytosolic increases of calcium at early and late stages of apoptosis [64, 70, 117]. In the lab of Dr. DeCoster, we have just begun to explore the calcium dynamics in multiple cell types with treatments of staurosporine, glutamate, potassium chloride (KCl), and ionomycin. We are working with several different cell lines – astrocytes, glioblastoma and Jurkat T cells. Additionally, the DeCoster lab is establishing its own primary astrocyte cell line of an adult rat brain. Using multiple cell types will grant us a deeper understanding into the underlying mechanisms. Preliminary data generated in the lab is very interesting, as glutamate appears to delay apoptosis in astrocytes, yet has no effect in T cells. It is unsurprising that the Jurkat T cells show no response to glutamate stimulation, since they lack both glutamate transporters and receptors. However, it would be beneficial to explore the reasons why glutamate delays apoptosis in astrocytes.

It seems, calcium is able to exhibit both pro- and anti-apoptotic tendencies. In Figure 6.2, we provide an overall picture of possible reactions that we can add to the model described in Chapter 4 (Figure 4.1). We can attempt to derive the reaction kinetics using the existing literature, and use experimental results from the lab of Dr. DeCoster to decipher the unknowns. Below, we elaborate on the details from the
literature for the inspiration behind Figure 6.2.

Ca\textsuperscript{2+} has a unique ability to establish local concentrations in the cytoplasm. The physical and chemical aspects of Ca\textsuperscript{2+}, which enable localization, allow different functionality based on the spatiotemporal dynamics of this signaling molecule. Moreover, due to its low diffusion rate – 100-fold lower than other typical second messengers, cyclic AMP and inositol 3-phosphate (IP3) – and tendency to become sequestered by different organelles in the cell, intracellular Ca\textsuperscript{2+} is tightly controlled during the cell’s lifespan [93].

Figure 6.2 The picture represents how calcium relates to the proteins of the Fas-mediated apoptotic signaling cascade.

Calcium release from the Endoplasmic Reticulum (ER) can have profound impact on the mitochondria and, thus, the intrinsic apoptotic pathway. The “cross-talk”
between the ER and the mitochondria is responsible for mediating signaling, ATP production and apoptosis [119]. The functional significance of the physical and physiological link between the ER and mitochondria is profound [39, 104]. For instance, in many apoptotic models the release of Ca\(^{2+}\) from the ER is directly responsible for mitochondrial calcium overload [40, 92]. Although Ca\(^{2+}\) has a low affinity for the mitochondrial Ca\(^{2+}\) transporters, it is the close proximity between the mitochondria and the ER which allows for the rapid accumulation of calcium in the mitochondrial matrix [91]. The interactions between Ca\(^{2+}\) and the mitochondria, can lead to a variety of mitochondrial activity. For instance, Ca\(^{2+}\) overload can result in a loss of mitochondrial membrane potential, which can lead to increased release of Cytochrome c into the cytosol.

Specifically, we are very interested in the dynamics of Ca\(^{2+}\), IP\(_3\)R and IP\(_3\), and their interactions in a negative feedback loop proposed in [12]. In the aforementioned paper, the authors were able to show a reduction in apoptotic behavior in Fas ligand-stimulated cells, where the Cytochrome c/IP\(_3\)R binding was inhibited. Alternatively, cells lacking IP\(_3\)R activity (knockout models and genetic deletions) show resistance to apoptosis [10, 54, 59, 115]. Other models have been suggested in which Bcl-2 family members interact with the type 1 version of IP\(_3\)R to control Ca\(^{2+}\) flow from the ER – e.g., Bax, Bad, and Bcl-X\(_L\) [85, 125]. Investigation into the role of IP\(_3\)R in calcium signaling may reveal a new pharmacological target in apoptosis [93]. Although calcium-related oscillations have been previously investigated (first in [76]), they have not yet been explored computationally within the context of Fas-mediated apoptosis.
Anti-apoptotic molecules in the cell have been shown to reduce cytosolic calcium levels when they are upregulated. For instance, Bcl-2 and Ca\(^{2+}\) have been shown to relate to each other. In [4] the authors observed Bcl-2 overexpression prevents Ca\(^{2+}\) release from the ER. The authors of [66] were able to show a reduction in the release of Ca\(^{2+}\) from the ER in relation to Bcl-2 levels.

Pro-apoptotic molecules in relation to calcium release from the ER have also been studied. The double knockout embryonic fibroblasts in [23], where pro-apoptotic molecules Bax and Bak were deleted, had a major reduction in levels of calcium in the ER. The double knockout models were resistant to a variety of apoptotic stimuli [108]. In addition, silencing Bcl-2 in the double knockout models partially restored the ER calcium levels. Finally, the green tea compound epigallocatechin gallate reduced the leakage of calcium from the ER and restored ER calcium levels in Bcl-2 overexpressing cells – N.B., the compound is known to bind to Bcl-2 and deactivate the protein. Hence, it interplay between elements in Figure 4.1 and Figure 6.2 is clear, and investigation into these phenomena can lead to new insight into cancers, autoimmune and neurodegenerate disorders.

Finally, as a tertiary research direction, we have the option of investigating calcium oscillations in relation to HIV latency. There is recent evidence supporting the idea that calpains may play a critical role in apoptosis. These cysteine proteases are activated in a Ca\(^{2+}\)-dependent manner, and they may be effective in inhibiting the activation of latent HIV-infected cells [116]. Hence, as we establish our models of Ca\(^{2+}\)-mediated apoptotic signaling, we can also work the new dynamics from the lab of Dr. DeCoster into the HIV apoptotic model we proposed in 5.
6.3 Modifying NWT

The NWT algorithm could be improved by parallelizing the algorithm. To date, the use of MPI has been restricted to running multiple simulations at the same time. We have used MPI to initialize the same system on multiple nodes, to generate multiple simulation results faster. There is another way to consider using parallel computing facilities to speed up the algorithm and increase the number of nondeterministic decisions made throughout a simulation run. The idea would involve partitioning the system into multiple subsystems, and having these subsystems interact through diffusion-like reactions. Consider the very simple system illustrated in Figure 6.3. N.B., there is a lot of whitespace in Figure 6.3, because we will require more space in Figure 6.4.

![Figure 6.3 A proposed reaction network for the partitioned system.](image)

The system in Figure 6.3 describes a few reactions working on a few proteins. Now, let us assume that all of the molecules of the system are accounted for in Figure
6.4. If we ignore the lines for a moment, then we see the system will initialize with six molecules of $A$, nine molecules of $B$, eight molecules of $C$, and five molecules of $D$. Now, looking at the lines, we can consider dividing the system into four Membrane Systems, each one simulating on its own node, but communicating by allowing molecules to float from one system to the next.

![Diagram](image)

Figure 6.4 Partition the cell into subcompartments for parallel simulation.

In Figure 6.4 we can see four different groups. Each group would be initialized as a Membrane System. We would need to develop communication laws to send a protein from one Membrane System to the next. Therein lies the difficulty. We would need to develop the mathematical framework behind when to send a protein from one Membrane System to the next – some sort of diffusion law. By breaking the system up into smaller subsystems, we can utilize the benefits of multiple processors, but we can also open opportunities for more nondeterministic decisions.
Our nondeterminism comes in the form of reaction competition, so it would be beneficial (nondeterministically speaking) for us to divide the system up in this manner. With less molecules in each Membrane System, the possibility for nondeterminism increases. The reaction memory would also play a larger role, since we would expect reactions per subsystem to have infinite Waiting Times more often, given less reactant molecules per subsystem. From Figure 6.4 it appears that subsystem III is the only system with a nucleus. If we are dividing the system to deal with the spatio characteristics of proteins, then this would make sense. Perhaps a more appropriate way to divide the system, would be to give each subsystem a piece of every compartment. See Figure 6.5.

![Diagram](image)

**Figure 6.5** A different view at breaking the system into subcompartments.

In Figure 6.5 we see a way to break up the system such that every rule has a chance to be implemented. Now every rule has a chance to execute each subsystem.
Some systems will come to lack certain reactants while other systems will have an abundance of those reactants. Hence, nondeterminism may play a greater role, as there will be a greater chance for reaction competition per subsystem. To implement this type of extension to the NWT algorithm we need to be concerned with the following:

1. How many subsystems do we define? What would be the logic behind defining the total number of subsystems?

2. How do we decide to initialize the protein multiplicities per subsystem? Do we merely divide the total number of molecules of each protein by the number of subsystems?

3. How do we send proteins between compartments? A simple rule to send one to another would suffice, but what sort of kinetics do we associate to these rules? One would think sending one protein to a new subsystem should be governed by more than mere random number generation.

There are some questions to be answered on this sort of implementation. As stated, this sort of setup would give more weight to the nondeterminism and memory enhancement described in this work. If a mathematical framework could be developed for breaking the system into subsystems, the logic should also be transferable to applying a similar partitioning scheme to the Gillespie Algorithm.

6.4 The Future of Systems Biology

It is interesting to consider the future of computational and systems biology. The field(s) are contingent upon the growth and production of so many areas of science.
At the heart of this new wave of collaborative efforts sweeping the Earth is the field of Systems Biology. The types of scientists involved in the field -- biologists, chemists, physicists, computer scientists, mathematicians, biochemists, biophysicists, neuroscientists, and so on -- is evergrowing, as humanity plunges deeper into solving the mysteries of ourselves.

Not to overshadow the events surrounding the discovery of the structure of DNA, the sequencing methods of Sanger, and other great biological or chemical discoveries, but the twentieth century seemed dominated by what physics could accomplish. We are leaving the era where humankind attempted to destroy itself with two great wars. We are entering an era filled with incredible biological possibilities. We see discoveries on genes which astound us -- the obese gene, the gay gene, etc. These discoveries are a bit short-sighted, in the sense that there are probably more factors than one gene which governs behavior. Nevertheless, we are beginning to understand the possibilities of our manipulation of genetic factors.

My interest in the future of this field lies in defining the specifics of the interaction networks. Whole cell simulation, while a very nice notion, seems quite far away. We are not equipped with the proper biochemical knowledge. We have exact methods for biochemical simulation, but these are essentially intractable for large reaction networks.

Current estimates put the number of possible interactions in a whole cell in the 100,000 range. Indeed, many of the simulations in this work had millions of interactions occur. To simulate an entire cell with realistic (stochastic) results, the amount of computational power required would be extraordinarily. I look forward to the fu-
ture of computer science and systems biology, as computers grow ever faster and our knowledge of the biochemistry of life continues to develop.
APPENDIX A

MATLAB CODE FOR

CIRCADIAN RHYTHM MODEL
function sol = simpp53;
	sol = ode23(@simpp53,[0 400],[1 1 0 0 0 0 0 0]);

figure
yplot=sol.y;
plot(sol.x,yplot(8,:),'LineWidth',3);
h = legend('ODE Results',15);
xlabel('Time','fontsize',18);
ylabel('Molecules','fontsize',18);
axis([0 400 0 3000]);

function dydt = simpp53(t,y,Z);
Aa=50;
Aaa=500;
Ar=0.01;
Arr=50;
Ba=50;
Br=5;
dMa=10;
dMr=0.5;
da=1;
dr=0.05;
ga=1;
gr=1;
gc=2;
Oa=50;
Or=100;

dydt = [0a*y(3)-ga*y(1)*y(6)
Or*y(4)-gr*y(2)*y(6)
 ga*y(1)*y(6)-0a*y(3)
 gr*y(2)*y(6)-Or*y(4)
 Aaa*y(3)+Aa*y(1)-dMa*y(5)
 Ba*y(5)+Oa*y(3)+Or*y(4)-y(6)*(ga*y(1)+gr*y(2)+gc*y(8)+da)
 Arr*y(4)+Ar*y(2)-dMr*y(7)
 Br*y(7)-gc*y(6)*y(8)+da*y(9)-dr*y(8)
 gc*y(6)*y(8)-da*y(9)
 ];
APPENDIX B

SOURCE CODE NWT.C
Cascader.c vl.7
by John Jack

Synopsis: A discrete nondeterministic simulation technique based
on Membrane Systems, developed by John Jack and Andrei Paun

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include "cascader.h"
//BEGIN MPI BLOCK
#ifdef USE_MPI
#include "mpi.h"
#endif
//END MPI BLOCK

//********** GLOBAL VARIABLE DECLARATIONS ******************
//Alphabet tail and head
int alphabet_tail;
int rules_tail;

//Create the heap (which points to the reaction list)
struct reaction *heap[RULES];

//An array to keep track of Lagged WTs
#ifdef USE_LAGS
double tau1lags[MAXLAGS];
double tau2lags[MAXLAGS];
int tau1head,tau2head;
int tau1tail,tau2tail;
#endif

//Debugging variable used to stop the program if there is an error
int stop; //Used in program

//************************ MAIN METHOD *************************

* The main() method has several features:
* (i) calls function importSBML to read information and build
* membrane system (protein alphabet[], reaction rules[])
* (ii) Initialize the Membrane System (evolution sim[])  
* (iii) Initializes the heap (BuildMinHeap)  
* (iv) Launch User Interface Menu  
**/

```c
int main(int argc, char *argv[]) {

//BEGIN MPI BLOCK
#ifdef USE_MPI

int numprocs, rank, namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
//Initialize the stuff
MPI_Init(&argc, &argv);
//Report number of processes in communicator comm
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
//Report rank, a number between 0 and count-1,
//identifying the calling process
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
//Useful for debugging -- returns name of processor
//on which it was called
MPI_Get_processor_name(processor_name, &namelen);
#endif
//END MPI BLOCK

//Variables for building the MEMBRANE SYSTEM
//Struct for maintaining all the information on the alphabet
// or proteins of the system
struct protein alphabet[ALPHABET];
//There are no proteins in the system yet (to be read
//from the xml file)
alphabet_tail = 0;
//Struct for maintaining all the information on the
//reaction/rules of the system
struct reaction rules[RULES];
//There are no rules in the system yet (to be read from the
//xml file)
rules_tail = 0;

//Set it true
stop = ISFALSE;

//Test and read the file (given as cmdline parameter) to
//initialize the simulator (populate alphabet[] and rules[])
importSBML(rules,alphabet,argv[1]);
//Initialize the simulator
initializeMembraneSystem(rules,alphabet);
```
//Initially build the minheap
buildMinHeap(heap);

/*
* Below is the UI, a simple user interface for launching the
* simulation run
*/

//BEGIN NONMPI BLOCK
#endif USE_MPI
int choice;
do{
    printf("*************** MAIN MENU ***************\n" );
    printf(" 1: Begin Simulation (NWT Algorithm)\n" );
    printf(" 2: Begin Simulation (Gillespie)\n" );
    printf(" 3: Print Initial WTs\n" );
    printf(" 4: Print Rules (reactions)\n" );
    printf(" 5: Print Kinetics\n" );
    printf(" 6: Print Alphabet (proteins)\n" );
    printf(" 7: Print Reactionlist for each protein\n" );
    printf(" 8: Find Max Reaction List\n" );
    printf(" 9: Quit\n" );
    printf("Command me: ");
    scanf("%d", &choice);
    switch(choice) //CHANGES FIX MENU
    {
    case 1:
        //NWT Algorithm
        evolver(rules, alphabet, heap);
        break;
    case 2:
        //Gillespie SSA
        gillesp(rules,alphabet);
        break;
    case 3:
        //Print the initial WTs
        printWTs(heap);
        break;
    case 4:
        //Print the list of rules
        printRules(rules,alphabet);
        break;
    case 5:
        //Print the discrete rate constants for all rules
        printKinetics(rules,alphabet);
break;
case 6:
    // Print the initial multiplicities
    printAlphabet(alphabet);
    break;
case 7:
    // Print the reaction dependences (reactants and products)
    printReactionlist(rules, alphabet, heap);
    break;
case 8:
    findMaxReactionlist(rules, alphabet);
    break;
case 9:
    break;
default:
    printf("NOT VALID ENTRY!\n");
}
}while(choice != 9);
printf("Cascader has been terminated by user\n");
#endif
// END NONMPI BLOCK

// BEGIN MPI BLOCK
#ifdef USE_MPI
#ifdef GA
    // Run simulator
    evolver(rules, alphabet, heap, rank);
#endif
#ifdef GA
    gillesp(rules, alphabet, rank);
#endif
MPI_Finalize();
#endif
// END MPI BLOCK
// return 0;
}//end method main()

/*
 * Method importSBML() reads in all of the information to initialize
 * the membrane system from the file, which was passed to the program
 * as a cmdline parameter. All of the information is read into the
 * alphabet[] and rules[] arrays. Also, the array of pointers to rules
 * *heap[] array is built having the array of pointers to elements of
 * rules[] allows us to build and maintain a heap without destroying
void importSBML(struct reaction *rules, struct protein *alphabet, char *sbml_file)
{
  //Needed for XML parser
  FILE *fp;
  //Read String
  char str[READBUFF];
  char xmlcase[10] = "NOTHING";
  //Temp str
  char *strtemppoint; //Used in XML
  char *strnpos; //Position of the substring
  char strtemp[100]; //Used for XML
  char *strtemp2[100]; //Used to interpret the double for kinetic XML
  double concentration; //Used to convert conc. into multiplicity
  int temp; //A temp (used for reaction XML parsing
  //Debugging for XML read DEBUG
  int numOfCompartments;
  int numOfReactions;
  int numOfProteins;
  char *tag = NULL;
  //Counter
  int c;

  //************** READ IN XML FILE **************
  //Check cmdline arg for filename and attempt to open with read privs
  if((fp = fopen(sbml_file, "r")) == NULL)
  {
    printf("ERROR: You must provide the SBML code to initialize "+
           "the simulator\n");
    printf("%%%%%%%%%%%%%%%%%%%%%%**************\\n");
    printf("USAGE: ./a.out <FILENAME>.xml.\\n");
    printf("e.g., ./a.out sample.xml.\\n");
    exit(1);
  }
  while(!feof(fp))
  {
    //READ STRING
    if(fgets(str, READBUFF, fp))
    {
      //**********CHECKING XML TAGS**********
    }
  }
}
// Check title
if (strstr(str,"<model id=") != NULL)
{
    strtok(str,"\"";
    printf("Successfully opened file %s\n",sbml_file);
    printf("Reading from project %s\n",strtok(NULL,"\""));
}

// Find Compartments
if (strstr(str,"<compartment id=") != NULL)
{
    numOfCompartments++;
    strtok(str,"\"";
    printf("Compartment %s\n",strtok(NULL,"\""));
}

/************** READ ALPHABET ********************/

// Start reading proteins
if (strstr(str,"<listOfSpecies>") != NULL)
{
    strcpy(xmlcase,"proteins");
}

// Stop reading in Proteins
if (strstr(str,"</listOfSpecies>") != NULL)
{
    strcpy(xmlcase,"NOTHING");
}

// Find Information on Each Protein
if ((strcmp(xmlcase,"proteins") == 0) &&
    (strstr(str,"<species ") != NULL))
{
    // Read in a protein
    // Increment tail to read in a new protein
    alphabet_tail++;
    // Get id for current reaction
    // Create temp of string so strtok does not cut original to pieces
    strtemp=str;
    strpos = strstr(strtemp,"id=");
    // Store location of first character for id string
    strtok(strpos,"\\n");
    // Must copy strtok to temp pointer then copy to string
    // Else SegFaul
    strtemppoint = strtok(NULL,"\\n");
    // Save ID
    strcpy(alphabet[alphabet_tail].id,strtemppoint);

    // Get Name for current reaction
//Create temp of string so strtok does not cut original to pieces
strcpy(strtemp,str);
strpos = strstr(strtemp,"name=");
strtok(strpos,$"$);
strtemppoint = strtok(NULL,$"$);
//strcpy(strtemp,strtemppoint);
//Save Name
strcpy(alphabet[alphabet_tail].name,strtemppoint);

//Get compartment for current reaction
//Create temp of string so strtok does not cut original to pieces
strcpy(strtemp,str);
strpos = strstr(strtemp,"compartment=");
strtok(strpos,$"$);
strtemppoint = strtok(NULL,$"$);
//Save Compartment
strcpy(alphabet[alphabet_tail].compartment,strtemppoint);

//Get concentration for current reaction
//Create temp of string so strtok does not cut original to pieces
strcpy(strtemp,str);
strpos = strstr(strtemp,"initialAmount=");
strtok(strpos,$"$);
strtemppoint = strtok(NULL,$"$);
//Save Concentration
concentration = atof(strtemppoint); //Store concentration temp
//Convert Initial Concentration into Initial # of Molecules
alphabet[alphabet_tail].multiplicity = concentration * AVO;
//Initialize the reactionlist_tail
alphabet[alphabet_tail].reactionlist_tail = 0;
}

//Start reading proteins
if (strstr(str,"<listOfReactions>") != NULL)
{
strcpy(xmlcase,"reactions");
}

//Stop reading proteins
if (strstr(str,"</listOfReactions>") != NULL)
{
strcpy(xmlcase,"NOTHING");
}

readcr //*************** READ REACTIONS ***************
//Find the reaction, put it in the array
if ((strcmp(xmlcase,"reactions") == 0) &&
(strstr(str,"<reaction ") != NULL))
{ //Read in a reaction //Increment tail to read in a new reaction rules_tail++; strtok(str,""); strtemppoint = strtok(NULL,""); //Save ID of reaction strcpy(rules[rules_tail].id,strtemppoint); //Initialize fields in struct rules[rules_tail].reactants_tail = 0; rules[rules_tail].products_tail = 0; //Inialize the nondeterministic counters rules[rules_tail].nondetermin = 0; rules[rules_tail].no_nondetermin = 0; //Make heap element point to reaction heap[rules_tail] = &rules[rules_tail]; (*heap[rules_tail]).heap_index = rules_tail; }

//Find the reactants for current reaction if (strstr(str,"<listOfReactants>") != NULL)
{
    strcpy(xmlcase,"reactants");
}

//Store reactant information if ((strcmp(xmlcase,"reactants") == 0) &&
    (strstr(str,"<speciesReference>") != NULL))
{
    //Read in a reactant
    //Get reactant Id
    strtok(str,"");
    strtemppoint = strtok(NULL,"");
    strcpy(strtemp, strtemppoint);
    //Find the reactant in alphabet and store its alphabet index
    for (c = 1; c <= alphabet_tail; C++)
        if (strcmp(strtemp,alphabet[c].id) == 0)
            { //Store index in alphabet array for the reactant 
                rules[rules_tail].reactants_tail++; 
                temp = rules[rules_tail].reactants_tail; 
                rules[rules_tail].reactants[temp] = c; 
                //Store reaction index
                alphabet[c].reactionlist_tail++; 
                alphabet[c].reactionlist[alphabet[c].reactionlist_tail] = rules_tail; 
            } 
} 
}
//End the reactantlist
if (strstr(str, "</listOfReactants>") != NULL)
{
    strcpy(xmlcase,"reactions");
}
//Find the products for current reaction
if (strstr(str, "<listOfProducts>") != NULL)
{
    strcpy(xmlcase,"products");
}
//Store reactant information
if ((strcmp(xmlcase,"products") == 0) &&
    (strstr(str,"<speciesReference>") != NULL))
{
    //Get reactant Id
    strtok(str,"\\""); strtemppoint = strtok(NULL,"\\"");
    strcpy(strtemp, strtemppoint);
    //Find the reactant in alphabet and store its alphabet index
    for (c = 1; c <= alphabet_tail; c++)
    if (strcmp(strtemp,alphabet[c].id) == 0)
    {
        //Store index in alphabet array for the product
        rules[rules_tail].products_tail++; temp = rules[rules_tail].products_tail;
        rules[rules_tail].products[temp] = c;
    }
}
//End the productlist
if (strstr(str, "</listOfProducts>") != NULL)
{
    strcpy(xmlcase,"reactions");
}
//Read in kinetics
if (strstr(str, "<listOfParameters>") != NULL)
{
    strcpy(xmlcase,"parameters");
}
if ((strcmp(xmlcase,"parameters") == 0) &&
    (strstr(str,"<parameter>") != NULL))
{
    strtok(str,"\\"");
    strtok(NULL,"\\"");
    strtok(NULL,"\\""); strtemppoint = strtok(NULL,"\\"");
strcpy(strtemp, strtemppoint);
rules[rules_tail].kinetic = atof(strtemp);
}
//End kinetics
if (strstr(str,"</listOfParameters>") != NULL)
{
strcpy(xmlcase,"reactions");
}
}
}//END XML PARSING WHILE LOOP
//Close file read
fclose(fp);
printf("Finished reading XML file\n");
}//end method importSBML

/*
 * The method initializeMembraneSystem() launches three methods
 * (i) Constrs must be calculate before WTs as they are used in WT
 * calculation
 * (ii) Initialize lag merely ensures all lags are set to 0 until
 * user can identify the lags (if any)
 */
void initializeMembraneSystem(struct reaction *rules,
struct protein *alphabet)
{
initializeConstr(rules);
initializeWTs(rules, alphabet);
#ifdef USE_LAGS
initializeLag(rules);
#endif
}//end method initializeMembraneSystem

//**** PRINTING METHODS **********//
/*
 * The method printAlphabet() does just that. It the entire
 * alphabet (proteins) to the screen
 * Output: Protein <name>, <id>, <compartment>, <current_multiplicity>
 */
void printAlphabet(struct protein *alphabet)
{
printf("*************************
Printing Alphabet...
");
int c;
for (c = 1; c <= alphabet_tail; c++)
{
printf("%d: Protein %s (%s), %s, %d molecules\n",}
c,alphabet[c].name,alphabet[c].id, 
alphabet[c].compartment,alphabet[c].multiplicity); 
} } //end method printAlphabet

/*
* The method printRules() does just that. It the entire set of rules 
* (reactions) to the screen 
* NOTE: The rules are read from rules[], which means they are always 
* printed in the same order as they were read in from SBML 
* So, the current WT does not affect the order (unlike *heap[]) 
* Output: <rules_index>: Reaction <name>, <constr>, <kinetic_rate>, 
* <current_WT>, <lag>
*/ 
void printRules(struct reaction *rules, struct protein *alphabet) 
{ 
printf("***************************
Printing Rules...
"n");
int c,i;
for (c = 1; c <= rules_tail; C++)
{
#ifdef USE_LAGS
printf("%d:Reaction %s: constr = %lf, k = %lf, WT = %lf, " +
"lag = %lf
REACTANTS: ",c,rules[c].id,rules[c].constr,
rules[c].kinetic,rules[c].waitingtime,rules[c].lag);
#endif
#ifdef USE_LAGS
printf("%d:Reaction %s: constr = %lf, k = %lf, WT = %lf, " +
lag = %lf
REACTANTS: ",c,rules[c].id,rules[c].constr,
rules[c].kinetic,rules[c].waitingtime);
#endif
for (i = 1; i <= rules[c].reactants_tail; i++)
printf("%s ",alphabet[rules[c].reactants[i]].name);
printf("\nPRODUCTS: ");
for (i = 1; i <= rules[c].products_tail; i++)
printf("%s ",alphabet[rules[c].products[i]].name);
printf("\nThere are %d reactants and %d products\n", 
rules[c].reactants_tail,rules[c].products_tail);
} } //end method printRules

/*
* The method printKinetics() does just that. It the kinetics per rule 
* (or reaction) to the screen (mostly for debugging and make sure 
* Membrane System is initialized properly.
void printKinetics(struct reaction *rules, struct protein *alphabet) {
    printf("***************************\nPrinting Rules...\n");
    int c,i;
    for (c = 1; c <= rules_tail; c++) {
        #ifdef USE_LAGS
            printf("%d:Reaction %s: constr = */.lf, k = */.lf, WT = */.lf, "
                    +"Lag = */.lf\n",c,rules[c].id,rules[c].constr,
                    rules[c].kinetic,rules[c].waitingtime,rules[c].lag);
        #endif
        #ifndef USE_LAGS
            printf("%d:Reaction */.s: constr = */.lf, k = */.lf, WT = */.lf, "+"Lag = */.lf\n",c,rules[c].id,rules[c].constr,
                    rules[c].kinetic,rules[c].waitingtime);
        #endif
    }
}

/*
The method printReactionlist() does just that.
* Print the list of reactions of which each protein is a part.
*/
void printReactionlist(struct reaction *rules, struct protein *alphabet, struct reaction **local_heap) {
    printf("***************************\nPrinting Reactionlists...\n");
    int c,i; //counters
    int tempindex; //store reactionlist tail for each protein
    int temp2;
    for (c = 1; c <= alphabet_tail; c++) {
        //For simplicity
        tempindex = alphabet[c].reactionlist_tail;
        printf("Protein */.s C/.s) : ",
                alphabet[c].name,alphabet[c].id);
        for (i = 1; i <= tempindex; i++) {
            //Get index in heap of reaction
            temp2 = rules[alphabet[c].reactionlist[i]].heap_index;
            printf("%s ",(*local_heap[temp2]).id);
        }
        printf("\n");
    }
The method printWTs() does just that. Prints the Reactions and WTs in the order of the minHeap

```c
void printWTs(struct reaction **local_heap)
{
    printf("***************************\nPrinting WTs...
");
    int c, i;
    for (c = 1; c <= rules_tail; c++)
    {
        printf("Reaction %s: WT=%lf and Mem=%lf\n", (*local_heap[c]).id,
(*local_heap[c]).waitingtime,(*local_heap[c]).memory_perc);
    }
} //end method printWTs
```

The method evolve() is called when the user chooses
* to start the simulator. The initializeSim() method is called
* to prompt user for number of cycles (seconds) to run the
* simulation. And, evolve() carries out the simulation until
* desired simulation time is reached, or terminates if ALL
* molecules are exhausted -- i.e., all WTs are inf
* does just that. It prints the Reactions and WTs in the
* order of the minHeap
*
```
BEGIN MPI BLOCK
#else USE_MPI
void evolve(struct reaction *rules, struct protein *alphabet,
struct reaction **local_heap, int rank)
END MPI BLOCK
#endif
```
//Initialize the values of sim
//BEGIN MPI BLOCK
#ifdef USE_MPI
initializeSim(&sim,rank);
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
initializeSim(&sim);
#endif
//END NONMPI BLOCK

//The tie
int ties_tail = 0;
//The multiplicities
int multiplicities_tail = 0;
//Information counters -- keep track of simulator workings
int nondeterministic_count = 0;
int random_count = 0;
int ties_count = 0;

//Create a tie heap, (check for multiple reactions with minWT)
int ties[MAXREACTIONS];
//Create a 2D array to store the previous
//multiplicities per cycle of proteins
int multiplicities[ALPHABET][PREVIOUSMULT];

//BEGIN MPI BLOCK
#ifdef USE_MPI
//Export the initial conditions (multiplicities at time t=0)
exportInitialMultiplicity(alphabet, &multiplicities_tail,rank);
#endif
//END MPI BLOCK

//BEGIN NONMPI BLOCK
#ifndef USE_MPI
//Export the initial conditions (multiplicities at time t=0)
exportInitialMultiplicity(alphabet, &multiplicities_tail);
#endif
//END NONMPI BLOCK

//Time the program
time_t start, finish;
//Start timer
time(&start);
//BEGIN MPI BLOCK
#ifdef USE_MPI
printInitialStepTo
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
printInitialStepTo
#endif
//END NONMPI BLOCK
while ((sim.completed_cycles < sim.goal_cycles) && (stop == ISFALSE))
{
    //Check to see if ALL rules are beyond the goal time
    //i.e., no rule can be applied
    if ((*local_heap[1]).waitingtime > sim.goal_cycles + 1)
    {
        printf("ALL RULES EXCEED THE GOAL CYCLES\n");
        printf("min WT = %lf and \nsimtime = %lf\n", (*local_heap[1]).waitingtime, sim.goal_cycles, sim.simulation_time);
        storeMultiplicity(alphabet, &multiplicities_tail, multiplicities);
        printf("Exporting at time t=\n", sim.simulation_time);
    }

    //BEGIN MPI BLOCK
#ifdef USE_MPI
exportMultiplicity(alphabet, &multiplicities_tail, multiplicities,rank);
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
exportMultiplicity(alphabet, &multiplicities_tail, multiplicities);
#endif
//END NONMPI BLOCK
/*CHECK FOR INFINITY*/
if ((*local_heap[1]).waitingtime == (1.0/0.0))
    printf("ERROR: MODEL OUT OF MOLECULES (MIN WT IS INFINITY)\n");
        break;
    }
    //Simulator can apply rule
    else
    {
        //BEGIN MPI BLOCK
#ifdef USE_MPI
applyRule(rules, alphabet, local_heap, &sim, &ties_tail, ties,
#endif
//END MPI BLOCK
&nondeterministic_count, &random_count, &ties_count, rank);
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
applyRule(rules, alphabet, local_heap, &sim, &ties_tail, ties,
&nondeterministic_count, &random_count, &ties_count);
#endif
//END NONMPI BLOCK

//A second has completed
if (sim.simulation_time / (sim.completed_cycles + 1) > 1)
{
  //Aggregate completecycles
  sim.completed_cycles++;
  //Print current second on screen
  printf("%d\n", sim.completed_cycles);
  storeMultiplicity(alphabet, &multiplicities_tail, multiplicities);
  //See if multiplicity arrays are full
  if (multiplicities_tail == PREVIOUSMULT)
  {
    printf("Exporting at time t=%.lf\n", sim.simulation_time);
    //BEGIN MPI BLOCK
    #ifdef USE_MPI
    exportMultiplicity(alphabet, &multiplicities_tail, multiplicities, rank);
    #endif
    //END MPI BLOCK
    //BEGIN NONMPI BLOCK
    #ifndef USE_MPI
    exportMultiplicity(alphabet, &multiplicities_tail, multiplicities);
    #endif
    //END NONMPI BLOCK
  }
}
//BEGIN MPI BLOCK
#ifdef USE_MPI
printStepToFile(local_heap,sim.simulation_time, rank);
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifdef USE_MPI
printStepToFile(local_heap,sim.simulation_time);
#endif
printf("Storing final multiplicity at time = %.lf\n",
    sim.simulation_time);

BEGIN MPI BLOCK
#ifdef USE_MPI
//Export the final multiplicity
exportMultiplicity(alphabet, &multiplicities_tail,
multiplicities, rank);
//Extraneous information for analysis and debugging postrun
printFinalInfo(rules,alphabet, nondeterministic_count,
    random_count, ties_count,rank);
//Finish counting the time of the program
    time(&finish);
    printf("******************************************************\n");
    printf("* Simulator for rank %d finished in %.2f seconds *
", rank,difftime(finish, start));
    printf("******************************************************\n");
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
//Export the final multiplicity
exportMultiplicity(alphabet, &multiplicities_tail, multiplicities);
//Extraneous information for analysis and debugging postrun
printFinalInfo(rules,alphabet, nondeterministic_count, random_count, ties_count);
//Finish counting the time of the program
    time(&finish);
    printf("******************************************************\n");
    printf("* Simulator finished in %.2f seconds *
", difftime(finish, start));
    printf("******************************************************\n");
#endif
//END NONMPI BLOCK
} //end method evolver

/*
* The method initializeSim prepares the evolution struct
* [created in method evolver()] for simulation
*/
//BEGIN MPI BLOCK
#ifdef USE_MPI

void initializeSim(struct evolution *sim, int rank)
#endif
//END MPI BLOCK
//BEGIN MPI BLOCK
#ifdef USE_MPI
void initializeSim(struct evolution *sim)
#endif
//END MPI BLOCK
{
  //Initialize the timing for the simulator
  sim->completed_cycles = 0;
  sim->rules_applied = 0;
  sim->simulation_time = 0.0;
  sim->export_counter = 0;
  //BEGIN MPI BLOCK
  #ifdef USE_MPI
  sim->goal_cycles = 1000;
  #endif
  //END MPI BLOCK

  //BEGIN NONMPI BLOCK
  #ifndef USE_MPI
  //Read in the desired simulation runtime
  printf("Enter desired simulation time (seconds): ");
  scanf("%d", &sim->goal_cycles);
  printf("Simulating for %d seconds\n", sim->goal_cycles);
  #endif
  //END NONMPI BLOCK

  //Initialize random Number Generator
  #ifdef USE_MPI
  unsigned int iseed = (unsigned int)time(NULL)*rank;
  #endif
  #ifndef USE_MPI
  unsigned int iseed = (unsigned int)time(NULL);
  #endif
  srand(iseed);
}  //end method initializeSim

/*
 * The method applyRule() does just that.
 * The rule with the min WT is rules[0]. We refer to it as the
 * applied rule. Rules must be passed to method so it can be
 * passed to updateRules for referencing the "pointers" in struct
protein of the reactionlist. The method method evolver is invoked when the user choosing to start the simulator.

* //BEGIN MPI BLOCK
#endif
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void applyRule(struct reaction *rules, struct protein *alphabet, struct reaction **local_heap, struct evolution *local_sim, int *ties_tail, int ties[MAXREACTIONS], int *nondeterministic_count, int *random_count, int *ties_count, int rank)
//END NONMPI BLOCK
{
//Temp so we can recalculate heap[l], but keep track of reactants
struct reaction applied = (*local_heap[l]);
//Replace simulation time with the applied rule WT. Aggregate simtime
local_sim->simulation_time = applied.waitingtime;
//Store the head node in the reaction ties list
ties[0] = 1;
*ties_tail = 1;
//Recursively check for ties
checkTie(1, local_heap, ties_tail, ties);
//Counters
int i,c;

//There is no tie apply the rule at the top of the heap
if (*ties_tail == 1)
{
//Update multiplicities of reactants and products for applied rule
//Reactants multiplicity
for (i = 1; i <= (*local_heap[1]).reactants_tail; i++)
{
alphabet[(*local_heap[1]).reactants[i]].multiplicity--;
}
//Products multiplicity
for (i = 1; i <= (*local_heap[1]).products_tail; i++)
{
alphabet[(*local_heap[1]).products[i]].multiplicity++;
}
// For statistical purposes, aggregate total # of rules applied
local_sim->rules_applied++;

// and # of times particular rule is applied
(*local_heap[i]).times_applied++;

// Recalculate the WT for the Applied rule
recalcAppliedWT(alphabet, local_heap, local_sim);

// Heap maintenance after recalculating the WT of the top node
fixHeap(1, local_heap);

// Update all the rules affected by the applied rule
updateRules(applied, rules, alphabet, local_heap, local_sim);
}

// Test are at least two ties slotted to occur at the same moment
else
{
    printf("there is a tie");
    int enoughReactants = checkReactants(alphabet, local_heap,
ties_tail, ties);
    ties_count++;
    // Build an array of pointers to the reactions
    // So that we can update all the reactions affected by the tied
    // reactions. This is necessary to allow the heap to update
    // after each waiting time calculation. Otherwise stuff moves
    // ties no longer points to the reactions.
    struct reaction *tieptrs[MAXREACTIONS];
    for (i = 0; i < *ties_tail; i++)
    {
        tieptrs[i] = &(*local_heap[ties[i]]);
    }

    // Apply all tied reactions
    if (enoughReactants == ITRUE)
    {
        // Apply all the rules in the tie
        for (i = 0; i < *ties_tail; i++)
        {
            // Update multiplicities of reactants and products for applied rule
            // Reactants multiplicity
            for (c = 1; c <= (*local_heap[ties[i]]).reactants_tail; c++)
            {
                alphabet[(*local_heap[ties[i]]).reactants[c]].multiplicity--;
            }

            // Products multiplicity
            for (c = 1; c <= (*local_heap[ties[i]]).products_tail; c++)
            {
                alphabet[(*local_heap[ties[i]]).products[c]].multiplicity++;
            }
        }
    }
}  //For statistical purposes, aggregate total # of rules applied
local_sim->rules_applied++;
//and # of times particular rule is applied
(*local_heap[ties[i]]).times_applied++;
}
//Update all rules affected by the tie
for (i = 0; i < *ties_tail; i++)
{
    //Recalculate the wt for the Applied rule must point to ruleslist
    //because stuff has possibly moved
    recalcTiedWT((*tieptrs[i]).heap_index, alphabet, local_heap, local_sim);
    //Heap maintenance after recalculating the WT of the top node
    fixHeap((*tieptrs[i]).heap_index, local_heap);
    //Update all the rules affected by the applied rule
    updateRules((*tieptrs[i]).rules, alphabet, local_heap, local_sim);
}
}
//Nondeterministically apply as many tied reactions as possible
else
{
    //Build an array of pointers to the applied rules, so that we
    //can properly update the heap later
    struct reaction *applied_rules[MAXREACTIONS];
    int applied_rules_tail = 0;
    int random = 0;  //Used to pick the rule for application
    int temp_tail = *ties_tail;
    int apply_rule = ISFALSE;
    //A temporary variable to be used in swapping elements of the array
    struct reaction *temp;

    //Apply all the rules before updating the WTs of everything. We
    //update the WT for the applied rules, but only to help sort the heap
    for (i = 0; i < *ties_tail; i++)
    {
        //Check if there is only one option
        if (i == *ties_tail - 1)
        {
            random = 0;
        }
        //If more than one, generate random choice for array
        else
        {
            //Generate random int from 0 to ties_tail-1
random = rand() % temp_tail;
}
random_count++;

//See if enough reactants exist to apply rule
apply_rule = checkRule(random,alphabet, local_heap, ties);

//Enough reactants available
if (apply_rule == ISTRUE) {
    nondeterministic_count++;
    (*heap[(*tieptrs[random]).heap_index]).nondetermin++;
    (*heap[(*tieptrs[random]).heap_index]).times_applied++;
    //Update multiplicities of reactants and products for applied rule
    //Reactants multiplicity
    for (c = 1; c <= (*tieptrs[random]).reactants_tail; c++) {
        alphabet[(*tieptrs[random]).reactants[c]].multiplicity--;
    }
    //Products multiplicity
    for (c = 1; c <= (*tieptrs[random]).products_tail; c++) {
        alphabet[(*tieptrs[random]).products[c]].multiplicity++;
    }
    applied_rules[applied_rules_tail] = tieptrs[random];
    //Move to next position in the applied_rules array
    applied_rules_tail++;
    //Eliminate applied rule from the tie array
    temp = tieptrs[random];
    tieptrs[random] = tieptrs[temp_tail-1];
    tieptrs[temp_tail-1] = temp;
    //Decrease the number of rules still need applying
    temp_tail--;
} //Not enough reactants available (remove node, but do not apply)
else {
    (*heap[(*tieptrs[random]).heap_index]).no_nondetermin++;
    temp = tieptrs[random];
    tieptrs[random] = tieptrs[temp_tail-1];
    tieptrs[temp_tail-1] = temp;
    temp_tail--;
}
}
// Now fix the WT for each tied rule and ALL affected rules
for (i = 0; i < *ties_tail; i++)
{
  // Recalculate the wt for the Applied rule must point to ruleslist
  // because stuff has possibly moved
  recalcTiedWT((*tietptrs[i]).heap_index, alphabet, local_heap, local_sim);
  // Heap maintenance after recalculating the WT of the top node
  fixHeap((*tietptrs[i]).heap_index, local_heap);
  // Update all the rules affected by the applied rule
  updateRules((*tietptrs[i]), rules, alphabet, local_heap, local_sim);
}
*ties_tail = 0;
} // end method applyRule

/*
 * The method checkTie() checks to see if multiple reactions have
 * the minimum waitingtime in the heap. It stores the index
 * (from *heap[]) of all of the reactions with a tie
 */
void checkTie(int parent, struct reaction **local_heap,
int *ties_tail, int ties[MAXREACTIONS])
{
  // Check left child exists
  if (2 * parent <= rules_tail)
  {
    if (fabs((*local_heap[1]).waitingtime -
            (*local_heap[2 * parent]).waitingtime) < TOL)
    {
      // Store index for tie
      ties[*ties_tail] = (2 * parent);
      // Increase the tail index (so sim knows tie)
      (*ties_tail)++;
      // Call checkTie (check children of the tied node)
      checkTie(2 * parent, local_heap, ties_tail, ties);
    }
  }
  // Check right child exists
  if (2 * parent + 1 <= rules_tail)
  {
    if (fabs((*local_heap[1]).waitingtime -
            (*local_heap[2 * parent + 1]).waitingtime) < TOL)
    {
      // Store index for tie
      ties[*ties_tail] = (2 * parent + 1);
      // Increase the tail index (so sim knows tie)
      (*ties_tail)++;
      // Call checkTie (check children of the tied node)
      checkTie(2 * parent + 1, local_heap, ties_tail, ties);
    }
  }
}
//Store index for tie
ties[*ties_tail] = (2 * parent + 1);
//Increase the tail index (so sim knows tie)
(*ties_tail)++;
//Call checkTie (check children of the tied node
checkTie(2 * parent + 1, local_heap, ties_tail, ties);
}
}
}//end method checkTie

/*
* The method checkRule() checks to see if the tied rule
* (at *heap[index]) can be applied
* NOTE: it is only needed if there are any reactions trying
* to apply at the same exact time as the reaction
* with min WT (top of heap)
*/
int checkRule(int index, struct protein *alphabet,
struct reaction **local_heap, int ties[MAXREACTIONS])
{
    //Counter
    int c;
    //For each reactant of the tying reaction
    for (c = 1; c <= (*local_heap[ties[index]]).reactants_tail; c++)
    {
        if (alphabet[(*local_heap[ties[index]]).reactants[c]].multiplicity < 1)
            return ISFALSE;
    }
    return ISTRUE;
}//end method checkRule

/*
* The method checkReactants checks all of the reactants
* for all of the tied reactions to see if they can all be applied.
* If they can't it returns ISFALSE, otherwise it returns ISTRUE
*/
int checkReactants(struct protein *alphabet,
struct reaction **local_heap, int *ties_tail, int ties[MAXREACTIONS])
{
    //Temp for the reactants list
    int reactantmult[ALPHABET];
    //Counters
    int i,c;
    int temp;
//Initialize array
for (i = 1; i <= alphabet_tail; i++)
reactantmult[i] = 0;

//Keep track of how many reactants are needed to
//satisfy ALL reactions in tie
//For each reaction in the tie
for (i = 0; i < *ties_tail; i++)
{
//For each reactant of the tying reaction
for (c = 1; c <= (*local_heap[ties[i]]).reactants_tail; c++)
{
reactantmult[(*local_heap[ties[i]]).reactants[c]]++;
}
}

//Now we see if there are enough reactants for the equations
//For each protein in the alphabet
for (i = 1; i <= alphabet_tail; i++)
{
if (alphabet[i].multiplicity - reactantmult[i] < 0)
{
return ISFALSE;
}
}
return ISTRUE;
}//end method checkReactants

/*
 * The method updateRules() is called by the applied rule method
 * when there was no ties for WT. All rules affected by the
 * multiplicities changes from the applied rule must be recalculated.
 */
void updateRules(struct reaction applied, struct reaction *rules,
struct protein *alphabet, struct reaction **local_heap,
struct evolution *local_sim)
{
//N.B., the elements in k = alphabet[i].reactionlist[j] is the
//index of the associated reactions in the original list rules[k].
//The reaction is found in the heap by accessing
//(heap[rules[k].heap_index]). To save time and space, I do not
//simplify these accesses with temp variables.

//Counters
int i,c;
//Recalculate all reactions involving each reactant of the applied rule
for (i = 1; i <= applied.reactants_tail; i++)
{
    for (c = 1; c <= alphabet[applied.reactants[i]].reactionlist_tail; c++)
    {
        //Recalculate WT of the affected rule
        recalcAffectedWT(
            rules[alphabet[applied.reactants[i]].reactionlist[c]].heap_index,
            alphabet, local_heap, local_sim);
        //Heap maintenance after changing a WT
        fixHeap(
            rules[alphabet[applied.reactants[i]].reactionlist[c]].heap_index,
            local_heap);
    }
}

//Recalculate all reactions involving each product of the applied rule
for (i = 1; i <= applied.products_tail; i++)
{
    for (c = 1; c <= alphabet[applied.products[i]].reactionlist_tail; c++)
    {
        //Recalculate WT of the affected rule
        recalcAffectedWT(
            rules[alphabet[applied.products[i]].reactionlist[c]].heap_index,
            alphabet, local_heap, local_sim);
        //Heap maintenance after changing a WT
        fixHeap(
            rules[alphabet[applied.products[i]].reactionlist[c]].heap_index,
            local_heap);
    }
}

};//end method updateRules

;/*
 * After a cycle (second) is completed, we need to store the
 * multiplicity of each protein. The array is
 * multiplicities[proteinindex][time].
 */
void storeMultiplicity(struct protein *alphabet,
              int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT])
{
    //Counter
    int i;
for (i = 1; i <= alphabet_tail; i++)
{
    //Store the multiplicity at the time
    multiplicities[i][*multiplicities_tail] = alphabet[i].multiplicity;
}
(*multiplicities_tail)++;
//end method storeMultiplicity

/ *
* The method exportMultiplicity() is used to export the
* multiplicities of all proteins for all of the simulation
* timepoints (usually seconds). It is not to be confused
* with the method exportInitialMultiplicity which is called
* only once to output the multiplicities at time t=0.
* This method can be called multiple times in a simulation run,
* because the multiplicities array (for each protein) has a
* max = MAXMULT, which is defined in the cascader.h file.
* /
//BEGIN MPI BLOCK
#ifdef USE_MPI
void exportMultiplicity(struct protein *alphabet,
    int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT],
    int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void exportMultiplicity(struct protein *alphabet,
    int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT])
#endif
//END NONMPI BLOCK
{
    //BEGIN MPI BLOCK
#ifdef USE_MPI
    //Convert the rank to a string so we can create appropriate output
    //file
    char srank[100];
    sprintf(srank, "results%o.csv", rank+1);
    FILE *ofp;
    printf("Begin exporting Values for thread %d...\n",rank);
    printf("Opening existing file ('results%d.csv')\n",rank);
    //File output
    //Open file with write privs
    ofp = fopen(srank,"a");
    
    for (i = 1; i <= alphabet_tail; i++)
    {
        //Store the multiplicity at the time
        multiplicities[i][*multiplicities_tail] = alphabet[i].multiplicity;
    }
    (*multiplicities_tail)++;
    //end method storeMultiplicity

    / *
    * The method exportMultiplicity() is used to export the
    * multiplicities of all proteins for all of the simulation
    * timepoints (usually seconds). It is not to be confused
    * with the method exportInitialMultiplicity which is called
    * only once to output the multiplicities at time t=0.
    * This method can be called multiple times in a simulation run,
    * because the multiplicities array (for each protein) has a
    * max = MAXMULT, which is defined in the cascader.h file.
    * /
    */
//BEGIN MPI BLOCK
#ifdef USE_MPI
void exportMultiplicity(struct protein *alphabet,
    int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT],
    int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void exportMultiplicity(struct protein *alphabet,
    int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT])
#endif
//END NONMPI BLOCK
{
if (ofp == NULL)
{
    fprintf(stderr, "Can't open output file results.csv!\n");
    exit(1);
}
#endif
//END MPI BLOCK

//END NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
printf("Begin exporting Values...\n");
printf("Opening existing file ('results.csv')\n");
//File output
//Open file with write prives
ofp = fopen("results.csv","a");
//END NONMPI BLOCK
if (ofp == NULL)
{
    fprintf(stderr, "Can't open output file results.csv!\n");
    exit(1);
}
#endif

int i,c; //counters
for (c = 0; c < *multiplicities_tail; c++)
{
    for (i = 1; i <= alphabet_tail; i++)
    {
        if (i == alphabet_tail)
            fprintf(ofp, "%d\n",multiplicities[i][c]);
        else
            fprintf(ofp, "%d,",multiplicities[i][c]);
    }
}
fclose(ofp);
*multiplicities_tail = 0;

/*
 * The method exportInitialMultiplicity() is used to export
 * the multiplicities of all proteins for the time t=0.
 * It is called only once. All other exporting of multiplicities
 * is handled by the method exportMultiplicity()
/*

//BEGIN MPI BLOCK
#ifdef USE_MPI
void exportInitialMultiplicity(struct protein *alphabet,
   int *multiplicities_tail, int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void exportInitialMultiplicity(struct protein *alphabet,
   int *multiplicities_tail)
#endif
//END NONMPI BLOCK
{
//BEGIN MPI BLOCK
#ifdef USE_MPI

//Convert the rank to a string so we can create appropriate output
//file
char srank[100];
sprintf(srank,"results%o.csv",rank+1);

FILE *ofp;
printf("Begin exporting Values for thread %d...\n",rank);
printf("Opening existing file ('results%d.csv')\n",rank);
//File output
ofp = fopen(srank,"w"); //Open file with write prives
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
printf("Begin exporting Values...\n");
printf("Opening new file\n");
//File output
//Open file with write prives
ofp = fopen("results.csv","w");
#endif
//END NONMPI BLOCK

if (ofp == NULL)
{
   fprintf(stderr, "Can't open output file %s!\n", "results.csv");
   exit(1);
}
//Counters
int i,c;
for (i = 1; i <= alphabet_tail; i++)
{
  if (i == alphabet_tail)
    fprintf(ofp, "%d\n",alphabet[i].multiplicity);
  else
    fprintf(ofp, "%d",alphabet[i].multiplicity);
}
fclose(ofp);
*multiplicities_tail = 0;

//BEGIN MPI BLOCK
#ifdef USE_MPI
void printlnitialStepToFile(struct reaction **local_heap,
double simulation_time, int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void printlnitialStepToFile(struct reaction **local_heap,
double simulation_time)
#endif
{
  //BEGIN MPI BLOCK
#ifdef USE_MPI
    //convert the rank to a string so we can create appropriate output
    //file
    char srank[100];
    sprintf(srank,"simulation%o.csv",rank+1);
    FILE *ofp;
    //File output
    //Open file with write prives
    ofp = fopen(srank,"w");
#endif
  //END MPI BLOCK

  //BEGIN NONMPI BLOCK
#ifndef USE_MPI
    FILE *ofp;
    //Open file with write prives
    ofp = fopen("simulation.csv","w");
#endif
  //END NONMPI BLOCK
if (ofp == NULL)
{
fprintf(stderr, "Can’t open output file %s!\n","info.csv");
exit(1);
}

//Print the nondeterminism information
//Counters
int c,i;
fprintf(ofp,"Simulation Time =,lf\n",simulation_time);
fprintf(ofp,"Reaction ID,WT,Memory,Nondeterministic Applied,"+
Nondeterministic Not Applied,Total Applied\n");
for (c = 1; c <= rules_tail; c++)
{
//Print Information on each Rule
fprintf(ofp,"%s,%lf,%lf,%d,%d,%d\n",(*local_heap[c]).id,
(*local_heap[c]).waitingtime,(*local_heap[c]).memory_perc,
(*local_heap[c]).nondetermin,(*local_heap[c]).no_nondetermin,
(*local_heap[c]).times_applied);
}
//Close file output
fclose(ofp);
}//end method printlnitialStepToFile

//BEGIN MPI BLOCK
#ifdef USE_MPI
void printStepToFile(struct reaction **local_heap,
double simulation_time, int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void printStepToFile(struct reaction **local_heap,
double simulation_time)
#endif
{
//BEGIN MPI BLOCK
#ifdef USE_MPI
//Convert the rank to a string so we can create appropriate output
//file
char srank[100];
sprintf(srank,"simulation%o.csv",rank+1);
FILE *ofp;
//File output
//Open file with write privileges
ofp = fopen(srank,"a");
#endif
//END MPI BLOCK

//BEGIN NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
//Open file with write prives
ofp = fopen("simulation.csv","a");
#endif
//END NONMPI BLOCK
if (ofp == NULL)
{
fprintf(stderr, "Can't open output file %s!\n","info.csv");
exit(1);
}
//Print the nondeterminism information
int c,i;
//printf("What the Crap?");
fprintf(ofp,"Simulation Time =,%lf\n",simulation_time);
fprintf(ofp,"Reaction ID,WT,Memory,Nondeterministic Applied,"+
Nondeterministic Not Applied,Total Applied\n");
for (c = 1; c <= rules_tail; c++)
{
//Print Information on each Rule
fprintf(ofp,"%s,%lf,%lf,%d,%d,%d\n",(*local_heap[c]).id,
(*local_heap[c]).waitingtime,(*local_heap[c]).memory_perc,
(*local_heap[c]).nondetermin,(*local_heap[c]).no_nondetermin,
(*local_heap[c]).times_applied);
}
//Close file output
fclose(ofp);
}//end method printStepToFile

/ *
* The method printFinalInfo() is used to gain information on
* the simulator. Mostly, it targets the nondeterministic aspects
* of the simulator.
*/

//BEGIN MPI BLOCK
ifndef USE_MPI
void printFinalInfo(struct reaction *rules, struct protein *alphabet,
int nondeterministic_count, int random_count, int ties_count,
int rank)
#endif
//END MPI BLOCK
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void printFinalInfo(struct reaction *rules, struct protein *alphabet,
int nondeterministic_count, int random_count, int ties_count)
#endif
//END NONMPI BLOCK
{
//BEGIN MPI BLOCK
#ifdef USE_MPI
//Convert the rank to a string so we can create appropriate output
//file
char srank[100];
sprintf(srank,"info°/o.csv",rank+1);
FILE *ofp;
printf("Exporting final information for %d...
",rank);
//File output
//Open file with write prives
ofp = fopen(srank,"a");
#endif
//END MPI BLOCK

//BEGIN NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
printf("Exporting Final Information...
");
//Open file with write prives
ofp = fopen("info.csv","w");
#endif
//END NONMPI BLOCK

if (ofp == NULL)
{
fprintf(stderr, "Can't open output file %s!\n", "info.csv");
exit(1);
}
fprintf(ofp, "# of ties = %d\n",ties_count);
fprintf(ofp, "# of random variables generated = %d\n",random_count);
fprintf(ofp, "# of nondeterministically applied rules = %d\n",
nondeterministic_count);
//Print the nondeterminism information
//Counters
int c,i;
fprintf(ofp,"Index , ID , Nondeterministic Applied ,
Nondeterministic Not Applied, Total Applied, Reactant 1, "+
"Reactant 2, Reactant 3, Product 1, Product 2, Product 3\n); for (c = 1; c <= rules_tail; c++)
{
  //Print information on the number of times rule was applied
  //at the end of run
  fprintf(ofp,"%d,%s,%d,%d,%d",c, rules[c].id, rules[c].nondetermin,
  rules[c].no_nondetermin, rules[c].times_applied);
  //Print the Reactants for the reaction
  for (i = 1; i <= 3; i++)
  {
    if (i <= rules[c].reactants_tail)
      fprintf(ofp,"%s,".alphabet[rules[c].reactants[i]].name);
    else
      fprintf(ofp",",.alphabet[rules[c].reactants[i]].name);
  }
  //Print the products for the reaction
  for (i = 1; i <= 3; i++)
  {
    if (i <= rules[c].products_tail)
      fprintf(ofp,"%s,".alphabet[rules[c].products[i]].name);
    else
      fprintf(ofp",",.alphabet[rules[c].products[i]].name);
  }
  //End the line for the next reaction information
  fprintf(ofp,\n"");
}
//Close the file IO
fclose(ofp);
}//end method printFinalInfo

//************************** REACTION METHODS **************************//
*/
* The method initializeConstr() is run once before the user is
  * prompted with the initial menu. The Constrs are needed to
  * calculate the WTs for each reaction. Constr is based on the
  * kinetic rate, Avogadro's number, and the order of the reaction
  * Currently the simulator has a max order of three. If necessary,
  * all higher order reactions can be divided into subreactions of
  * lower order in the SBML file
*/
void initializeConstr(struct reaction *rules)
{
  //Counter
  int i;
  for (i = 1; i <= rules_tail; i++)
if (rules[i].reactants_tail == 2)
    rules[i].constr = rules[i].kinetic / AVO;
else if (rules[i].reactants_tail == 3)
    rules[i].constr = rules[i].kinetic / (AVO * AVO);
else
    rules[i].constr = rules[i].kinetic;
}//end method initializeConstr

/*
 * The method initializeLag() is run once before the user is prompted
 * with the initial menu. All lags are set to 0, and remain this way
 * unless the user specifies the lags.
 */
#ifdef USE_LAGS
void initializeLag(struct reaction *rules)
{
    //Counter
    int i;
    //Initialize lag values for all rules
    for (i = 1; i <= rules_tail; i++)
    {
        rules[i].lag = 0;
        rules[i].lagMultiplicity = 0;
        rules[i].tau = 0;
    }
    //Initialize the two lag arrays -- set all values to inf
    for (i = 0; i < MAXLAGS; i++)
    {
        tau1lags[i] = (1.0/0.0);
        tau2lags[i] = (1.0/0.0);
    }
    //Initialize the heads and tails for the queues
    tau1head = 0;
    tau2head = 0;
    tau1tail = 0;
    tau2tail = 0;
}//end method initializeLag
#endif

/*
 * The method initializeWTs() calculates the WT for each reaction
 * at t=0. It is run only once, which is before the user is prompted
 * with the menu.
 */
void initializeWTs(struct reaction *rules, struct protein *alphabet)
{
    printf("Initializing the WTs...
");
    int i;
    //rule_WT is a pointer to a reactions WT
    double *rule_WT;
    double *rule_Mem;
    double constr;
    #ifdef USE_LAGS
    double lag;
    #endif
    for (i = 1; i <= rules_tail; i++)
    {
        rule_WT = &rules[i].waitingtime;
        rule_Mem = &rules[i].memory_WT;
        rules[i].memory_perc = 1.0;
        constr = rules[i].constr;
        #ifdef USE_LAGS
        lag = rules[i].lag; //CHANGES
        calcInitialWTs(rule_WT,rule_Mem,constr,rules[i].reactants,
                        rules[i].reactants_tail,alphabet,lag);
        #endif
        #ifndef USE_LAGS
        calcInitialWTs(rule_WT,rule_Mem,constr,rules[i].reactants,
                        rules[i].reactants_tail,alphabet);
        #endif
    }
} //end method initializeWTs

void changeLag(struct reaction *rules, struct protein *alphabet)
{
    /**************SETUP TAU1************************************/
    //Print the rules so user can choose the two lag rules from list
    printRules(rules,alphabet);
    //Counters
    int c,i;
double templag;
printf("Which rule is tau1?\n");
scanf("%d", &c);
//Confirm Choice
printf("You have chosen %d\n",c);
//Print data for chosen rule
printf("%d: Reaction %s: constr = %lf, k = %lf, WT = %lf,"+
" Lag = %lf
REACTANTS: ",c, rules[c].id, rules[c].constr,
rules[c].kinetic, rules[c].waitingtime, rules[c].lag);
for (i = 1; i <= rules[c].reactants_tail; i++)
printf("%s ", alphabet[rules[c].reactants[i]].name);
printf("\nPRODUCTS: ");
for (i = 1; i <= rules[c].products_tail; i++)
printf("%s ", alphabet[rules[c].products[i]].name);
printf("\nThere are %d reactants and %d products\n", 
rules[c].reactants_tail, rules[c].products_tail);
templag = 0;
printf("What is the value of lag tau1?\n");
scanf("%lf", &templag);
rules[c].lag = templag;
printf("Lag tau1 has been changed to %lf\n", rules[c].lag);
//Tell the rule that it is tau1
//---> so it points to the right array, tau1lags[]
rules[c].tau = 1;

UILDER TAU2

**********SETUP TAU2********************
printf("Which rule is tau2?\n");
scanf("%d", &c);
//Confirm Choice
printf("You have chosen %d\n",c);
//Print data for chosen rule
printf("%d: Reaction %s: constr = %lf, k = %lf, WT = %lf,"+
" Lag = %lf
REACTANTS: ",c, rules[c].id, rules[c].constr,
rules[c].kinetic, rules[c].waitingtime, rules[c].lag);
for (i = 1; i <= rules[c].reactants_tail; i++)
printf("%s ", alphabet[rules[c].reactants[i]].name);
printf("\nPRODUCTS: ");
for (i = 1; i <= rules[c].products_tail; i++)
printf("%s ", alphabet[rules[c].products[i]].name);
printf("\nThere are %d reactants and %d products\n", 
rules[c].reactants_tail, rules[c].products_tail);
templag = 0;
printf("What is the value of lag tau2?\n");
scanf("%lf", &templag);
rules[c].lag = templag;
printf("Lag tau2 has been changed to °/lf\n", rules[c].lag);
//Tell the rule that it is tau1
//--> so it points to the right array, tau1ags[]
rules[c].tau = 2;
//Reinitialize WTs and heap
buildMinHeap(heap);
} //end method changeLag
#endif

/*
* The method calcInitialWTs() calculates the WT time t=0.
* It is run only once, which is before the user is prompted
* with the menu
*/
#ifdef USE_LAGS
void calcInitialWTs(double *rule_WT, double *rule_Mem, double constr, int reactants[], int reactants_tail, struct protein alphabet[], double lag)
#endif
#ifndef USE_LAGS
void calcInitialWTs(double *rule_WT, double *rule_Mem, double constr, int reactants[], int reactants_tail, struct protein alphabet[])
#endif
{
    //Counter
    int i;
    //Calculate the velocity based on kinetics
    double velocity = 1;
    for (i = 1; i <= reactants_tail; i++)
    {
        velocity = velocity * alphabet[reactants[i]].multiplicity;
    }
    velocity = velocity * constr;
    *rule_WT = (1 / velocity);
    *rule_Mem = (1 / velocity);
} //end method calcInitialWTs

/*
* The method recalcAppliedWT is called after the applied rule has
* been executed by simulator and the WT must be recalculated.
*/
void recalcAppliedWT(struct protein *alphabet,
struct reaction **local_heap, struct evolution *local_sim)
{
/Check if it is a lagging rule
#ifdef USE_LAGS
if ((*local_heap[1]).lag == 0)
{
#endif
/*BEGIN NON LAG VERSION*/
//Counter
int i;
//Calculate the velocity based on kinetics
double velocity = 1;
for (i = 1; i <= (*local_heap[1]).reactants_tail; i++)
{
velocity = velocity *
alphabet[(*local_heap[1]).reactants[i]].multiplicity;
}
velocity = velocity * (*local_heap[1]).constr;
(*local_heap[1]).waitingtime = (1 / velocity) +
local_sim->simulation_time;
(*local_heap[1]).memory_perc = 1.0;
/*END NON LAG VERSION*/
#ifdef USE_LAGS
}
//Rule has Lag
else
{
//Since the rule was just applied, WE KNOW that the tail must
//move forward, unlike the other calcWT methods
if ((*local_heap[1]).tau == 1)
{
taulhead++;
//Check if exceeded array size, and wrap around if needed
if (taulhead == MAXLAGS)
taulhead = 0;
//If there are no more in queue, set WT to infinity
if (taulhead == taultail)
{
/*DEBUG*/ printf("There are no more molecules to lag (taul)");
(*local_heap[1]).waitingtime = (1.0 / 0.0);
(*local_heap[1]).lagmultiplicity = 0;
}
else
{
(*local_heap[1]).waitingtime = tauillags[taulhead];
(*local_heap[1]).lagmultiplicity--;
Since the rule was just applied, WE KNOW that the tail must move forward, unlike the other calcWT methods.

```c
if ((*local_heap[l]).tau == 2)
{
    tau2head++;
    //Check if exceeded array size, and wrap around if needed
    if (tau2head == MAXLAGS)
        tau2head = 0;
    //If there are no more in queue, set WT to infinity
    if (tau2head == tau2tail)
    {
        /*DEBUG*/ printf("There are no more molecules to lag (tau2)");
        (*local_heap[l]).waitingtime = (1.0 / 0.0);
        (*local_heap[l]).lagmultiplicity = 0;
    }
    else
    {
        (*local_heap[l]).waitingtime = tau2lags[tau2head];
        (*local_heap[l]).lagmultiplicity--;
    }
}
#endif
/*DEBUG printf("WT is NOW %lf\n",(*heap[l]).waitingtime);*/
@end method recalcAppliedWT
```

The method recalcTiedWT is called when the rules in the tied list may have moved, so we must recalculate the applied rule but it may not be in the head node position. So, it appears similar to the recalcAppliedWT method, but it doesn’t go for the head node position.

```c
/*
 * The method recalcTiedWT is called when the rules in the tied list may have moved, so we must recalculate the applied rule but it may not be in the head node position. So, it appears similar to the recalcAppliedWT method, but it doesn’t go for the head node position.
 */
void recalcTiedWT(int index, struct protein *alphabet,
                   struct reaction **local_heap, struct evolution *local_sim)
{
    #ifdef USE_LAGS
    if (((*local_heap[index]).lag == 0)
    {
        #endif
    /*BEGIN NON LAG VERSION*/
    //Counter
    int i;
    //Calculate the velocity based on kinetics
```
double velocity = 1.0;
for (i = 1; i <= (*local_heap[index]).reactants_tail; i++)
{
    velocity = velocity *
        alphabet[(*local_heap[index]).reactants[i]].multiplicity;
}   
velocity = velocity * (*local_heap[index]).constr;
(*local_heap[index]).waitingtime = (1 / velocity) + local_sim->simulation_time;
(*local_heap[index]).memory_perc = 1.0;
/*END NON LAG VERSION*/
#endif USE_LAGS
}
//Rule has Lag
else
{
    //Since the rule was just applied, WE KNOW that the tail must
    //move forward, unlike the other calcWT
    if ((*local_heap[index]).tau == 1)
    {
        tau1head++;
        //Check if exceeded array size, and wrap around if needed
        if (tau1head == MAXLAGS)
            tau1head = 0;
        //If there are no more in queue, set WT to infinity
        if (tau1head == taultail)
        {
            /*DEBUG*/ printf("There are no more molecules to lag (tau1)");
            (*local_heap[index]).waitingtime = 1.0 / 0.0;
            (*local_heap[index]).lagmultiplicity = 0;
        }
    }
    else
    {
        (*local_heap[index]).waitingtime = tau1lags[tau1head];
        (*local_heap[index]).lagmultiplicity--;
    }
    //Since the rule was just applied, WE KNOW that the tail must
    //move forward, unlike the other calcWT methods
    if ((*local_heap[index]).tau == 2)
    {
        tau2head++;
        //Check if exceeded array size, and wrap around if needed
        if (tau2head == MAXLAGS)
            tau2head = 0;
//If there are no more in queue, set WT to infinity
if (tau2head == tau2tail)
{
    /*DEBUG*/ printf("There are no more molecules to lag (tau2)\n");
    (*local_heap[index]).waitingtime = (1.0 / 0.0);
    (*local_heap[index]).lagmultiplicity = 0;
}
else
{
    (*local_heap[index]).waitingtime = tau2lags[tau2head];
    (*local_heap[index]).lagmultiplicity--;
}
}
/* printf("WT is NOW %lf\n",(*heap[1]).waitingtime);*/
}//end method recalcAppliedWT

/*
The method recalcAffectedWTAfter applying a rule, all
affected rules (sharing reactants or products) must have
their WT recalculated, and the heap must be fixed after
each recalculation (handled in updateRules method)
*/
void recalcAffectedWT(int index, struct protein *alphabet,
struct reaction **local_heap, struct evolution *local_sim)
{
    #ifdef USE_LAGS
    if ((*local_heap[index]).lag == 0)
    {
    #endif
    //Counter
    int i;
    //Calculate the velocity based on kinetics
double velocity = 1;
    for (i = 1; i <= (*local_heap[index]).reactants_tail; i++)
    {
        velocity = velocity * 
        alphabet[(*local_heap[index]).reactants[i]].multiplicity;
    }
    velocity = velocity * (*local_heap[index]).constr;
    //If the new WT is infinity (no molecules available)
    if (velocity == 0.0)
    {
if (((*local_heap[index]).waitingtime == (1.0/0.0))
(*local_heap[index]).memory_perc = 1.0;
else
{
  // (amt_of_time_left_to_wait) / (total_time_to_wait)
  (*local_heap[index]).memory_perc = ((*local_heap[index]).waitingtime -
  local_sim->simulation_time) / (*local_heap[index]).memory_WT;
}
// Set to infinity
(*local_heap[index]).waitingtime = 1.0 / velocity;
}
// Elseif WT is smaller
else if (((*local_heap[index]).memory_perc*(l / velocity) +
  local_sim->simulation_time) < (*local_heap[index]).waitingtime)
{
  (*local_heap[index]).waitingtime =
  (*local_heap[index]).memory_perc*(1.0 / velocity) +
  local_sim->simulation_time;
  (*local_heap[index]).memory_WT = (1.0 / velocity);
}
// Note: else, WT is not smaller and there are enough molecules
// to execute reaction, so do nothing
#ifdef USE_LAGS
}
// Rule has Lag
else
{
  if (((*local_heap[index]).tau == 1)
  {
    // If we actually have another molecule to lag (due to the products
    // of this rule, the former if statement will succeed when the product
    // concentration changes -- but there are no new reactants
    if (alphabet[(*local_heap[index]).reactants[1]].multiplicity >
    (*local_heap[index]).lagmultiplicity)
    {
      taullags[taultail] = local_sim->simulation_time +
      (*local_heap[index]).lag;
      taultail++;
      // Check if exceeded array size
      if (taultail == MAXLAGS)
      taultail = 0;
      (*local_heap[index]).lagmultiplicity++;
      // Store FIRST lag value
      if (((*local_heap[index]).lagmultiplicity == 1)
      (*local_heap[index]).waitingtime = taullags[taulhead];
if ((*local_heap[index]).lagmultiplicity == MAXLAGS - 5)
{
printWTs(heap);
printAlphabet(alphabet);
printf("***********\nERROR: Out of lag array space for tau1\n");
exit(1);
}
}
if ((*local_heap[index]).tau == 2)
{
//printf("Made it inside tau 2\n");
//If we actually have another molecule to lag (due to the products
//of this rule, the former if statement will succeed when the product
//concentration changes -- but there are no new reactants
if (alphabet[(*local_heap[index]).reactants[1]].multiplicity >
(*local_heap[index]).lagmultiplicity)
{
tau21ags[tau2tail] = local_sim->simulation_time +
(*local_heap[index]).lag;
tau2tail++;
//Check if exceeded array size
if (tau2tail == MAXLAGS)
tau2tail = 0;
//Aggregate the lagmultiplicity, because that molecule is cued
//up in the lag array now
(*local_heap[index]).lagmultiplicity++;
//Store FIRST lag value
if ((*local_heap[index]).lagmultiplicity == 1)
(*local_heap[index]).waitingtime = tau21ags[tau2head];
}
if ((*local_heap[index]).lagmultiplicity == MAXLAGS - 5)
{
printWTs(heap);
printAlphabet(alphabet);
printf("***********\nERROR: Out of lag array space for tau2\n");
exit(1);
}
}
#endif
//end method recalcAffectedWT

//*************** STANDARD HEAP MAINTENANCE ***************//
/*
* Initially build the min heap so the smallest waiting time is the
* first element.
*/
void buildMinHeap(struct reaction **local_heap)
{
    //Counter
    int count;
    for(count = (rules_tail/2); count >= 1; count--)
        heapDown(count, local_heap);
} //end method buildMinHeap

//Check children of node to see if minheap property holds true
void heapDown(int node, struct reaction **local_heap)
{
    //Set smallest
    int min = node;
    //Get right index
    int right = 2 * node + 1;
    //Get left index
    int left = 2 * node;
    //Check if left child exists
    if (left <= rules_tail)
        if ((*local_heap[left]).waitingtime < (*local_heap[min]).waitingtime) {
            min = left;
        }
    //Check if right child exists
    if (right <= rules_tail) {
        if ((*local_heap[right]).waitingtime < (*local_heap[min]).waitingtime) {
            min = right;
        }
    }
    if (min != node) {
        swap(min, node, local_heap);
        heapDown(min, local_heap);
    }
} //end method heapDown

/*
* The Swap method swaps two nodes in the heap
*/
void swap(int a, int b, struct reaction **local_heap)
{
    struct reaction *temp;
temp = local_heap[a];
local_heap[a] = local_heap[b];
local_heap[b] = temp;
//Store the heap index in the reaction node,
//for reference of molecule reaction list
(*local_heap[a]).heap_index = a;
(*local_heap[b]).heap_index = b;
}//end method swap

//Readjust the heap for the node with changed WT
void fixHeap(int node, struct reaction **local_heap)
{
    //Check parent of node for min-heap property
    //Does parent exist?
    if ((node/2) >= 1)
        if ((*local_heap[node/2]).waitingtime >
            (*local_heap[node]).waitingtime)
        {
            swap(node/2,node, local_heap);
            fixHeap(node/2, local_heap);
        }
    //Check children of node for min-heap property
    //Set smallest
    int min = node;
    //Check if left child exists
    if ((2*node) <= rules_tail)
        if ((*local_heap[(2*node)]).waitingtime <
            (*local_heap[min]).waitingtime)
            min = (2*node);
    //Check if right child exists
    if ((2*node+1) <= rules_tail)
        if ((*local_heap[(2*node+1)]).waitingtime <
            (*local_heap[min]).waitingtime)
            min = (2*node+1);
    if (min != node)
        {
            swap(min,node, local_heap);
            fixHeap(min, local_heap);
        }
}//end method fixHeap*/

//***********************************************************************
// EXTRA METHODS ***********************************************************************
//Check the maximum of reactions for a given protein
void findMaxReactionlist(struct reaction *rules,
struct protein *alphabet)
{  
  //Counter  
  int i;  
  //The max number of reactions associated to protein  
  int max = 0;  
  //Alphabet index of max  
  int max_index = 0;  
  for (i = 1; i < alphabet_tail; i++)  
  {  
    printf("Protein %s has %d reactions\n",alphabet[i].id,alphabet[i].reactionlist_tail);  
    if (alphabet[i].reactionlist_tail > max)  
    {  
      max_index = i;  
      max = alphabet[i].reactionlist_tail;  
    }  
  }  
  printf("*************** ");  
  printf("Protein %s has %d reactions associated with it.\n",alphabet[max_index].id,max);  
} //end method findMaxReactionlist

//********** GILLESPIE METHODS ************//
/*  
* The method printGillesp is used to check the values of h,  
* a, a0, gtau, gmu to see if we are rocking and rolling.  
*/
void printGillespRules(struct reaction *rules,  
struct protein *alphabet)  
{  
  printf("***************************\nPrinting Rules...\n");  
  int c,i;  
  for (c = 1; c <= rules_tail; c++)  
  {  
    printf("%d:Reaction %s: a = %lf, constr = %lf, k = %lf\nREACTANTS: ",c,rules[c].id,rules[c].a,rules[c].constr,rules[c].kinetic);  
    for (i = 1; i <= rules[c].reactants_tail; i++)  
    {  
      printf("%s ",alphabet[rules[c].reactants[i]].name);  
    }  
    printf("\n");  
  }  
} //end method printReactionlist

/*  
* The method prints the A vals for Gillespie  
*/
void printGillespAvals(struct reaction *rules)
{
    printf("***************************
Printing A vals...
\n");
    int c;
    for (c = 1; c <= rules_tail; c++)
    {
        printf("%d:Reaction %s: a = %lf\n", c, rules[c].id, rules[c].a);
    }
}

/*
 * The gillespie is the implementation of the original Gillespie SSA
 */
#ifdef USE_MPI
void gillesp(struct reaction *rules, struct protein *alphabet, int rank)
//END MPI BLOCK
#endif
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
void gillesp(struct reaction *rules, struct protein *alphabet)
#endif
{
    //Struct for maintaining all the information regarding an actual
    //simulation run (simtimes, exporting times, etc.)
    struct evolution sim;
    //BEGIN MPI BLOCK
    #ifdef USE_MPI
    initializeSim(&sim,rank);
    #endif
    //END MPI BLOCK
    //BEGIN NONMPI BLOCK
    #ifndef USE_MPI
    initializeSim(&sim);
    #endif
    //END NONMPI BLOCK
    //a0 to be passed as pointer to various functions
    double a0;
    //Create a 2D array to store the previous multiplicities
    //per cycle of proteins
    int multiplicities[ALPHABET][PREVIOUSMULT];
    int multiplicities_tail = 0;

    //Initially calculate h_is and a_is and a0
    initializeGillesp(rules, alphabet, &a0);
//Export the initial multiplicity of each protein to file on HD
//BEGIN NONMPI BLOCK
#ifndef USE_MPI
exportGillespInitialMultiplicity(alphabet, &multiplicities_tail);
#endif
//END NONMPI BLOCK

#ifdef USE_MPI
exportGillespInitialMultiplicity(alphabet, &multiplicities_tail,rank);
#endif

//Initialize random Number Generator
unsigned int iseed = (unsigned int)time(NULL);
srand(iseed);

//Generate r1 and r2
double r1 = (rand()/(double)RAND_MAX+1));
double r2 = (rand()/(double)RAND_MAX+1));
//Calculate gtau (In function and uses a0 and r1) and
//gmu (do-loop, add up ai's until you equal or exceed r2*a0)
//gtau = what time to apply rule[gmu]
double gtau = calculateGtau(rules,alphabet,r1,a0);
//gmu = what rule to apply rule[gmu]
int gmu = calculateGmu(rules,alphabet,r2,a0);

//Time the program
time_t start, finish;
//Start timer
time(&start);

do
{
    //Apply rule[gmu], update the multiplicities of products and reactants
    applyRuleGillesp(rules,alphabet,gmu,&a0);
    //Fix all as,hs, and a0 THIS IS BAD!!!
    initializeGillesp(rules,alphabet,&a0);
    //Aggregate the simulation time
    sim.simulation_time += gtau;
    //Output the mutliplicities
    if (sim.simulation_time / (sim.completed_cycles + 1) > 1)
    //THEN a second has completed, so we must export
    {
        //Aggregate completecycles
        sim.completed_cycles++;
    }

}
//Print current second on screen
printf("%d\n", sim.completed_cycles);
storeMultiplicity(alphabet, &multiplicities_tail, multiplicities);
if (multiplicities_tail == PREVIOUSMULT)
  //THEN the multiplicities arrays is full
{
  printf("Exporting at time t = %lf\n", sim.simulation_time);
  //BEGIN NONMPI BLOCK
#ifndef USE_MPI
  //Export the multiplicity arrays
  exportGillespMultiplicity(alphabet, &multiplicities_tail, multiplicities);
#endif
  //END NONMPI BLOCK
  //BEGIN MPI BLOCK
#ifdef USE_MPI
  //Export the multiplicity arrays
  exportGillespMultiplicity(alphabet, &multiplicities_tail, multiplicities, rank);
#endif
  //END MPI BLOCK
}

//Generate new r1 and r2
r1 = (rand()/(double)RAND_MAX+1);
r2 = (rand()/(double)RAND_MAX+1);
//Calculate gtau (ln function and uses a0 and r1) and
//gmu (do-loop, add up ai's until you equal or exceed r2*a0)
gtau = calculateGtau(rules,alphabet,rl,aO);
gmu = calculateGmu(rules,alphabet,r2,a0);
}

while (sim.completed_cycles < (sim.goal_cycles));

//Export some debugging information
printGillespFinalInfo(rules,alphabet,sim,gtau,gmu,aO);
#ifndef USE_MPI
  //Export the final multiplicity
  exportGillespMultiplicity(alphabet, &multiplicities_tail, multiplicities);
#endif

#ifdef USE_MPI
  //Export the final multiplicity
  exportGillespMultiplicity(alphabet, &multiplicities_tail, multiplicities,rank);
#endif
#endif

time(&finish);
printf("***************************\n");
printf("* Simulator finished in %.2f seconds *\n",
difftime(finish, start));
printf("***************************\n");
}
/*
* The method initializeGillesp calculates all of the his, ais,
* and a0 for the first time step his use constr instead of kinetic
*/
void initializeGillesp(struct reaction *rules,
struct protein *alphabet, double *a0)
{
    //temp for h
double temph;
    //temp for a0
double tempa0 = 0;
    //Counters
    int c, i;

    for (c = 1; c <= rules_tail; c++)
    {
        temph = 1;
        //Calculate h=X1*...*Xn for all reactants involved in reaction
        for (i = 1; i <= rules[c].reactants_tail; i++)
        {
            temph = temph * (double)alphabet[rules[c].reactants[i]].multiplicity;
        }
        rules[c].h = temph;
        //Calculate a=h*c
        rules[c].a = rules[c].h * rules[c].constr;
        /*DEBUG if (rules[c].a < 0)
        {
            printf("ERROR: 'a' is negative\n");
            //printGillespRules(rules,alphabet);
            printf("The problematic rule is %d",c);
            for (i = 1; i <= rules[c].reactants_tail; i++)
            {
                temph = temph * alphabet[rules[c].reactants[i]].multiplicity;
            }
            //printGillespRules(rules,alphabet);
            exit(1);
        }
*/
    }
}
}*/
}

//Calculate a0
for (c = 1; c <= rules_tail; c++)
{
    tempa0 += rules[c].a;
}
//Store new a0 value
*a0 = tempa0;

/*
* The method calculateT_
* */
double calculateGtau(struct reaction *rules, struct protein *alphabet, double r1, double aO)
{
    //return the value of gtau
    return ((1/aO) * log(l/r1));
}

/*
* The method calculateGmu() returns the gmu value
*/
int calculateGmu(struct reaction *rules, struct protein *alphabet, double r2, double aO)
{
    //Counter
    int c = 0;
    double goal = r2 * aO;
    double sum = 0;
    //Find the index, c, of the array of rules whose a value is s.t.
    //sum ai's up to ac is >= r2*a0
    do
    { 
        c++;  //start at 1
        sum += rules[c].a;
    }while (sum <= goal);
    return c;
}

/*
* The method applyRuleGillesp() is similar to the method
* applyRule for the NWT algorithm. It updates the multiplities
* for the reactants and products of the rule[gm]
*/
void applyRuleGillesp(struct reaction *rules,
struct protein *alphabet, int gmu, double *a0)
{
    //Counter
    int i;
    double temp, olda;
    rules[gm].times_applied++;
    //Update Multiplicities of reactants and products and a's and h's
    //Reactants multiplicity
    for (i = 1; i <= rules[gm].reactants_tail; i++)
    {
        alphabet[rules[gm].reactants[i]].multiplicity--;
    }
    //Products multiplicity
    for (i = 1; i <= rules[gm].products_tail; i++)
    {
        alphabet[rules[gm].products[i]].multiplicity++;
    }
}

/*
 * The method `exportInitialMultiplicity()` is used to export the
 * multiplicities of all proteins for the time t=0. It is called
 * only once. All other exporting of multiplicities is handled by
 * the method `exportMultiplicity()`
 */
#endif
#endif
void exportGillespInitialMultiplicity(struct protein *alphabet,
    int *multiplicities_tail)
#endif
#endif
#ifdef USE_MPI
void exportGillespInitialMultiplicity(struct protein *alphabet,
    int *multiplicities_tail, int rank)
#endif
{
    //BEGIN MPI BLOCK
#ifdef USE_MPI
    //Convert the rank to a string so we can create appropriate output
    //file
    char srank[100];
    sprintf(srank,"resultsGillesp\%d.csv",rank+1);
    FILE *ofp;
    printf("Begin exporting Values for thread \%d...\n",rank);
    printf("Opening existing file ('resultsGillesp\%d.csv')\n",rank);
    //File output
//Open file with write privies
ofp = fopen(srank,"w");
#endif
//END MPI BLOCK

//BEGIN NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
printf("Begin exporting Values...\n");
printf("Opening new file\n");
//File output
//Open file with write privies
ofp = fopen("resultsGillesp.csv","w");
#endif
//END NONMPI BLOCK

if (ofp == NULL)
{
    fprintf(stderr, "Can't open output file %s!\n", "resultsGillesp.csv");
    exit(1);
}
int i,c; //counters
for (i = 1; i <= alphabet_tail; i++)
{
    if (i == alphabet_tail)
        fprintf(ofp, "%d\n", alphabet[i].multiplicity);
    else
        fprintf(ofp, "%d," ,alphabet[i].multiplicity);
}
fclose(ofp);
*multiplicities_tail = 0;
}//end method exportGillespInitialMultiplicity

#ifndef USE_MPI
void exportGillespMultiplicity(struct protein *alphabet,
int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT])
#endif
#ifdef USE_MPI
void exportGillespMultiplicity(struct protein *alphabet,
int *multiplicities_tail, int multiplicities[ALPHABET][PREVIOUSMULT],
int rank)
#endif
{
    //BEGIN MPI BLOCK
    #ifdef USE_MPI

    //END MPI BLOCK
    #ifndef USE_MPI

    //END NONMPI BLOCK
#endif

    //BEGIN NONMPI BLOCK
    #ifdef USE_MPI

    //END NONMPI BLOCK
#endif

/convert the rank to a string so we can create appropriate output
//file
char srank[100];
sprintf(srank,"resultsGillesp\o.csv",rank+1);

FILE *ofp;
printf("Begin exporting Values for thread \%d...\n",rank);
printf("Opening existing file ('resultsGillesp\%d.csv')\n",rank);
//File output
//Open file with write privs
ofp = fopen(srank,"a");
if (ofp == NULL)
{
    fprintf(stderr, "Can't open output file resultsGillesp.csv!\n");
    exit(1);
}
#endif
//END MPI BLOCK

//BEGIN NONMPI BLOCK
#ifndef USE_MPI
FILE *ofp;
printf("Begin exporting Values...\n");
printf("Opening existing file ('resultsGillesp.csv')\n");
//File output
//Open file with write privs
ofp = fopen("resultsGillesp.csv","a");
//END NONMPI BLOCK
if (ofp == NULL)
{
    fprintf(stderr, "Can't open output file results.csv!\n");
    exit(1);
}
#endif
//END NONMPI BLOCK

int i,c; //counters
for (c = 0; c < *multiplicities_tail; c++)
{
    for (i = 1; i <= alphabet_tail; i++)
    {
        if (i == alphabet_tail)
        {
            fprintf(ofp, "\%d\n",multiplicities[i][c]);
        }
        else
        {
            fprintf(ofp, "\%d," ,multiplicities[i][c]);
        }
    }
}
fclose(ofp);
*multiplicities_tail = 0;
}//end method exportGillespMultiplicity

/*
 * The method printGillespFinalInfo prints the final
 * information on the simulation
 */
void printGillespFinalInfo(struct reaction *rules,
struct protein *alphabet, struct evolution sim, double gtau, int gmu,
double a0)
{
printf("The simulator has terminated at t = %lf\n"
"Next gtau = %lf (t+gtau = %lf)\ngmu = %d\nna0 = %lf\n",
sim.simulation_time,gtau,sim.simulation_time+gtau,gmu,a0);
}//end method pringGillespDebug()
//*************** CONSTANT DECLARATION ******************//
//Max length for XML line
#define READBUFF 200
//Max length for protein name
#define PROTEINNAME 50
//Max length for protein Id
#define PROTEINID 5
//Max length for reaction Id
#define REACTIONID 5
//Max length for compartment id
#define COMPARTMENTNAME 8
//Max # of previous cycles to be kept track of per protein
#define PREVIOUSMULT 10000
//Max # of proteins
#define ALPHABET 100
//Max # of rules
#define RULES 200
//Max # of reactants per reaction
#define MAXREACTANTS 5
//Max # of products per reaction
#define MAXPRODUCTS 5
//Max # of reactions associated to a given protein
#define MAXREACTIONS 21
//Avogadro's constant * Cell volume 10^-13
#define AV0 602.21415
//Max # of characters representing number for desired
//simtime in seconds e.g., 10000 seconds = 5 chars
#define SIMTIME 10
//Boolean operator
#define ISTRUE 1
//Boolean operator
#define ISFALSE 0
//Maximum number of lags in queue
#define MAXLAGS 90000
//Tolerance for comparing double values.
#define TOL .0000000000000001

//*************** PROTEIN STRUCT **********************//
/*
 * Instantiation of struct represents one protein
 */
struct protein
{
    char compartment[COMPARTMENTNAME];
    int multiplicity;
char name[PROTEINNAME];
char id[PROTEINID];
int reactionlist[MAXREACTIONS];
int reactionlist_tail;
int multiplicities[PREVIOUSMULT];
}; // end struct protein

// *************** REACTIONS STRUCT ***********************

* Instantiation of struct represents one reaction
*/
struct reaction
{
  // Id of reaction
  char id[REACTIONID];
  // The kinetic law of the Reaction
  double kinetic;
  // The kinetic constant (calculated from kinetic rate)
  double constr;
  // The waiting time of the Reaction (+simulation time)
  double waitingtime;
  // The waiting time of the reaction (not in the context of simulation)
  double memory_WT;
  // The percent of time the rule has already waited
  double memory_perc;
  // Lag time
  #ifdef USE_LAGS
  double lag;
  // Keeps track of how many molecules have been queued, to be checked
  // against the number of molecules available for queueing
  // when recalcing WT
  int lagmultiplicity;
  // Which lag is it?
  int tau;
  #endif
  // The index in the reactions heap\n  // Needed for pointing from molecule to reactions, to be resorted
  int heap_index;
  // The number of times the rule has been applied during a simulation
  int times_applied; /*DEBUG*/
  // The number of times the rule was nondetermistically applied
  int nondetermin;
  // The number of times the rule was nondeterministically chosen,
  // but could not be applied
  int no_nondetermin;
//Lists for reactants and products of an instance of Reaction
int reactants[MAXREACTANTS];
int products[MAXPRODUCTS];
//Tail of reactants and products describes how many there are
int reactants_tail;
int products_tail;
double probability;

//**********Gillespie Variables**********/
double h;
double a;
}; //end struct reaction

/******************* TIME STRUCT ******************************************/
struct evolution
{
    int completed_cycles;
    int goal_cycles;
    int rules_applied;
    double simulation_time;
    int export_counter;
}; //end struct evolution

/***************** METHOD DECLARATIONS **********************************/
//Extra Methods
void printAlphabet(struct protein *);
void printRules(struct reaction *, struct protein *);
void printWTs(struct reaction **);
void printKinetics(struct reaction *, struct protein *);
void printReactionlist(struct reaction *, struct protein *,
    struct reaction **);

//Membrane System without Lags
//BEGIN MPI BLOCK
#ifdef USE_MPI
    void evolver(struct reaction *, struct protein *,
        struct reaction **, int);
    void exportMultiplicity(struct protein *, int *,
        int [ALPHABET][PREVIOUSMULT], int);
    void exportInitialMultiplicity(struct protein *, int *, int);
    void printFinalInfo(struct reaction *, struct protein *, int,
        int, int, int);
    void printlnitialStepToFile(struct reaction **, double, int);
    void printStepToFile(struct reaction **, double, int);
#endif

#ifndef USE_MPI

// END MPI BLOCK
// BEGIN NONMPI BLOCK
void evolver(struct reaction *, struct protein *, struct reaction **);
void exportMultiplicity(struct protein *, int *,
int [ALPHABET][PREVIOUSMULT]);
void exportInitialMultiplicity(struct protein *, int *);
void printFinalInfo(struct reaction *, struct protein *, int,
int, int);
void printInitialStepToFile(struct reaction **, double);
void printStepToFile(struct reaction **, double);
#endif
// END NONMPI BLOCK
#endif
// BEGIN MPI BLOCK
void importSBML(struct reaction *, struct protein *, char *);
void initializeMembraneSystem(struct reaction *, struct protein *);
void initializeSim(struct evolution *);
void applyRule(struct reaction *, struct protein *,
struct reaction **, struct evolution *, int *, int [MAXREACTIONS],
int *, int *, int *);
void updateRules(struct reaction, struct reaction *,
struct protein *, struct reaction **, struct evolution *);
void storeMultiplicity(struct protein *, int *,
int [ALPHABET][PREVIOUSMULT]);
void checkTie(int, struct reaction **, int *, int [MAXREACTIONS]);
int checkReactants(struct protein *, struct reaction **,
int *, int [MAXREACTIONS]);
int checkRule(int, struct protein *, struct reaction **,
int [MAXREACTIONS]);
void arraySwap(struct reaction *, int, int);

// Membrane System with Lags
void evolverlag(struct reaction *, struct protein *,
struct reaction **);

// Gillespie SSA methods
// BEGIN MPI BLOCK
#ifndef USE_MPI

void gillesp(struct reaction *, struct protein *);
void exportGillespInitialMultiplicity(struct protein *, int *);
void exportGillespMultiplicity(struct protein *, int *,
int [ALPHABET][PREVIOUSMULT]);
#endif
// END NONMPI BLOCK
// BEGIN MPI BLOCK
#ifdef USE_MPI
void gillesp(struct reaction *, struct protein *, int);
void exportGillespInitialMultiplicity(struct protein *, int *, int);
void exportGillespMultiplicity(struct protein *, int *,
int [ALPHABET][PREVIOUSMULT], int);
#endif

void initializeGillesp(struct reaction *, struct protein *,
double *r);
void printGillesp(struct reaction *, struct protein *);
void printGillespAvals(struct reaction *);
double calculateGtau(struct reaction *, struct protein *,
double, double);
int calculateGmu(struct reaction *, struct protein *, double, double);
void applyRuleGillesp(struct reaction *, struct protein *, int,
double *);
void printGillespFinalInfo(struct reaction *, struct protein *,
struct evolution, double, int, double);

//Reaction methods
void initializeConstr(struct reaction *);
void initializeWTs(struct reaction *, struct protein[]);
void initializeLag(struct reaction *);
void changeLag(struct reaction *, struct protein *);
#ifdef USE_LAGS
void calcInitialWTs(double *, double *, double, int[], int,
struct protein[], double);
#endif
#ifdef USE_LAGS
#endif

void recalcAppliedWT(struct protein *, struct reaction **,
struct evolution *);
void recalcTiedWT(int, struct protein *, struct reaction **,
struct evolution *);
void recalcAffectedWT(int, struct protein *, struct reaction **,
struct evolution *);

//Heap methods
void buildMinHeap(struct reaction **);
void heapDown(int, struct reaction **);
void swap(int, int, struct reaction **);
void fixHeap(int, struct reaction **);
// Other methods
void findMaxReactionList(struct reaction *, struct protein *);
APPENDIX D

FAS MEMBRANE SYSTEM
Our model consist of the following P system:

$$\Pi = (\Sigma, \mu, M_e, M_s, M_c, M_m, R_e, R_s, R_c, R_m)$$

where:

- $$\Sigma = \{FASL, FAS, FASC, FADD, FASC:FADD, FASC:FADD_2, FASC:FADD_3, FASC:FADD_2:CASP8, FASC:FADD_3:CASP8, \ldots\}$$
- $$L = \{e, s, c, m\}$$ We have four compartments - the outside environment ($e$), the cell surface ($s$), the cytoplasm ($c$), and the mitochondria ($m$).
- $$\mu = [s | c | m | c]_s$$
- $$M_e, M_s, M_c$$ and $$M_m$$ are the sets of multiplicities (protein multiplicity).
  - $$M_e = \{FASL^{1204}\}$$
  - $$M_s = \{FAS^{6022}\}$$
  - $$M_c = \{FADD^{10038}, CASP8^{20071}, FLIP^{48779}, CASP3^{12042}, Bid^{1055}, Bax^{50182}, XIAP^{18066}, Apaf^{60220}, ATP^{6022141}, CASP9^{12044}\}$$
  - $$M_m = \{Smac^{60220}, Cyto.c^{60220}, Bcl2^{45166}\}$$
- $$R_e, R_s, R_c$$ and $$R_m$$ are the sets of rules associated to each compartment.
  - $$R_e = \{r_1\}$$
  - $$R_s = \{r_2, r_4, r_6, r_8, r_{10}, r_{11}, r_{12}, r_{14}, r_{16}, r_{17}, r_{18}, r_{20}, r_{22}, r_{24}, r_{26}, r_{28}, r_{30}, r_{32}, r_{34}, r_{36}, r_{38}, r_{40}, r_{42}, r_{44}, r_{46}, r_{48}, r_{50}, r_{52}, r_{54}, r_{56}, r_{58}, r_{60}, r_{62}, r_{63}, r_{64}, r_{65}, r_{66}\}$$
  - $$R_c = \{r_3, r_5, r_7, r_9, r_{11}, r_{13}, r_{15}, r_{17}, r_{19}, r_{21}, r_{23}, r_{25}, r_{27}, r_{29}, r_{31}, r_{33}, r_{35}, r_{37}, r_{39}, r_{41}, r_{43}, r_{45}, r_{47}, r_{49}, r_{51}, r_{53}, r_{55}, r_{57}, r_{59}, r_{61}, r_{63}, r_{67}, r_{69}, r_{70}, r_{71}, r_{72}, r_{73}, r_{74}, r_{75}, r_{76}, r_{77}, r_{78}, r_{79}, r_{80}, r_{81}, r_{82}, r_{83}, r_{84}, r_{85}, r_{86}, r_{87}, r_{88}, r_{89}, r_{90}, r_{91}, r_{92}, r_{93}, r_{94}, r_{95}, r_{96}, r_{98}\}$$
  - $$R_m = \{r_{97}, r_{99}\}$$
We list the four sets of rules we simulated, which are based upon the apoptosis pathway. Each set incorporates a mechanism for Bcl2 involvement in the inhibition of apoptosis.

<table>
<thead>
<tr>
<th>Label</th>
<th>Rule</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>( FASL[FAS]_s \rightarrow [FASC]_s )</td>
<td>( k_{lf} )</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>([FASC]_s \rightarrow FASL[FASC]_s )</td>
<td>( k_{1r} )</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>( FASC[FADD]_c \rightarrow FASC:FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>( FASC:FADD]_c \rightarrow FASC[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_5 )</td>
<td>( FASC:FADD[FADD]_c \rightarrow FASC:FADD2]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_6 )</td>
<td>( FASC:FADD2]_c \rightarrow FASC:FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_7 )</td>
<td>( FASC:FADD3]_c \rightarrow FASC:FADD2]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_8 )</td>
<td>( FASC:FADD2]_c \rightarrow FASC:FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_9 )</td>
<td>( FASC:FADD2]_c \rightarrow FASC:FADD3]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{10} )</td>
<td>( FASC:FADD3]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{11} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{12} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{13} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{14} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{15} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{16} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{17} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD8[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{18} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{19} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{20} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{21} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{22} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{23} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>( r_{24} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2f} )</td>
</tr>
<tr>
<td>( r_{25} )</td>
<td>( FASC:FADD2]_c \rightarrow FADD82[FADD]_c )</td>
<td>( k_{2r} )</td>
</tr>
<tr>
<td>Label</td>
<td>Rule</td>
<td>Rate</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>$r_{26}$</td>
<td>$FASC : FADD_3 : FLIP[ ]_c \rightarrow FASC : FADD_3 [ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{27}$</td>
<td>$FASC : FADD_3 : CASP8[ CASP8 ]_c \rightarrow FASC : FADD_3 : CASP8_2 [ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{28}$</td>
<td>$FASC : FADD_3 : CASP8_2[ ]_c \rightarrow FASC : FADD_3 : CASP8 [ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{29}$</td>
<td>$FASC : FADD_3 : CASP8 [ FLIP ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{30}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP[ ]_c \rightarrow FASC : FADD_3 : CASP8 [ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{31}$</td>
<td>$FASC : FADD_3 : FLIP[ CASP8 ]_c \rightarrow FASC : FADD_3 : CASP8 [ CASP8 ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{32}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP[ ]_c \rightarrow FASC : FADD_3 : FLIP[ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{33}$</td>
<td>$FASC : FADD_3 : FLIP[ FLIP ]_c \rightarrow FASC : FADD_3 : FLIP_2 [ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{34}$</td>
<td>$FASC : FADD_3 : FLIP_2[ ]_c \rightarrow FASC : FADD_3 : FLIP[ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{35}$</td>
<td>$FASC : FADD_3 : CASP8_2 [ CASP8 ]_c \rightarrow FASC : FADD_3 : CASP8_2 [ CASP8 ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{36}$</td>
<td>$FASC : FADD_3 : CASP8 [ ]_c \rightarrow FASC : FADD_3 : CASP8_2 [ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{37}$</td>
<td>$FASC : FADD_3 : CASP8_2 [ FLIP ]_c \rightarrow FASC : FADD_3 : CASP8_2 : FLIP[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{38}$</td>
<td>$FASC : FADD_3 : CASP8_2 : FLIP[ ]_c \rightarrow FASC : FADD_3 : CASP8_2 [ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{39}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP[ CASP8 ]_c \rightarrow FASC : FADD_3 : CASP8_2 : FLIP[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{40}$</td>
<td>$FASC : FADD_3 : CASP8_2 : FLIP[ ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP[ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{41}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP[ FLIP ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP_2 [ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{42}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP_2[ ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP[ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{43}$</td>
<td>$FASC : FADD_3 : FLIP_2[ CASP8 ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP_2[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{44}$</td>
<td>$FASC : FADD_3 : CASP8 : FLIP_2 [ ]_c \rightarrow FASC : FADD_3 : CASP8 : FLIP_2[ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{45}$</td>
<td>$FASC : FADD_3 : FLIP_2[ FLIP ]_c \rightarrow FASC : FADD_3 : FLIP_3 [ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{46}$</td>
<td>$FASC : FADD_3 : FLIP_3[ ]_c \rightarrow FASC : FADD_3 : FLIP_2 [ FLIP ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{47}$</td>
<td>$FASC : FADD_2 [ CASP8 ]_c \rightarrow FASC : FADD_2 : CASP8[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>$r_{48}$</td>
<td>$FASC : FADD_2 : CASP8 [ ]_c \rightarrow FASC : FADD_2 [ CASP8 ]_c$</td>
<td>$k_{3r}$</td>
</tr>
<tr>
<td>$r_{49}$</td>
<td>$FASC : FADD_2 [ FLIP ]_c \rightarrow FASC : FADD_2 : FLIP[ ]_c$</td>
<td>$k_{3f}$</td>
</tr>
<tr>
<td>Label</td>
<td>Rule</td>
<td>Rate</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>r_{50}</td>
<td>FASC : FADD_2 : FLIP[c] \rightarrow FASC : FADD_2[ FLIP ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{51}</td>
<td>FASC : FADD_2 : CASP8[ CASP8 ]_c \rightarrow FASC : FADD_2 : CASP8_2[ ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{52}</td>
<td>FASC : FADD_2 : CASP8_2[ ]_c \rightarrow FASC : FADD_2 : CASP8[ CASP8 ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{53}</td>
<td>FASC : FADD_2 : CASP8[ FLIP ]_c \rightarrow FASC : FADD_2 : CASP8[ FLIP ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{54}</td>
<td>FANC : FADD_2 : CASP8[ FLIP ]_c \rightarrow FASC : FADD_2 : FLIP[ ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{55}</td>
<td>FASC : FADD_2 : FLIP[ CASP8 ]_c \rightarrow FASC : FADD_2 : CASP8[ FLIP ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{56}</td>
<td>FASC : FADD_2 : FLIP[ ]_c \rightarrow FASC : FADD_2 : FLIP[ CASP8 ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{57}</td>
<td>FASC : FADD_2 : FLIP[ ]_c \rightarrow FASC : FADD_2 : FLIP_2[ ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{58}</td>
<td>FASC : FADD_2 : FLIP_2[ ]_c \rightarrow FASC : FADD_2 : FLIP[ FLIP ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{59}</td>
<td>FASC : FADD_3 : CASP8_3[ ]_c \rightarrow FASC : FADD_3 : CASP8[ ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{60}</td>
<td>FANC : FADD_3 : CASP8[ ]_c \rightarrow FASC : FADD_3[ CASP8 ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{61}</td>
<td>FASC : FADD_3 : FLIP[ ]_c \rightarrow FASC : FADD_3 : FLIP[ ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{62}</td>
<td>FASC : FADD_3 : FLIP[ ]_c \rightarrow FASC : FADD_3[ FLIP ]_c</td>
<td>k_{3r}</td>
</tr>
<tr>
<td>r_{63}</td>
<td>FASC : FADD_3 : CASP8_2[ ]_c \rightarrow FASC : FADD_2[ CASP8 ]_c</td>
<td>k_{3f}</td>
</tr>
<tr>
<td>r_{64}</td>
<td>FASC : FADD_3 : CASP8_2[ ]_c \rightarrow FASC : FADD_3[ CASP8^2 ]_c</td>
<td>k_{4}</td>
</tr>
<tr>
<td>r_{65}</td>
<td>FASC : FADD_3 : CASP8_2[ ]_c \rightarrow FASC : FADD_3[ FLIP ]_c</td>
<td>k_{4}</td>
</tr>
<tr>
<td>r_{66}</td>
<td>FASC : FADD_3 : CASP8_2[ ]_c \rightarrow FASC : FADD_3[ CASP8 ]_c</td>
<td>k_{4}</td>
</tr>
<tr>
<td>r_{67}</td>
<td>CASP8^2_2[ ]_c \rightarrow CASP8^2_2[ ]_c</td>
<td>k_{5}</td>
</tr>
<tr>
<td>r_{68}</td>
<td>CASP8^2_2[ , CASP3 ]_c \rightarrow CASP8^2_2[ , CASP3 ]_c</td>
<td>k_{6r}</td>
</tr>
<tr>
<td>r_{69}</td>
<td>CASP8^2_2[ CASP3 ]_c \rightarrow CASP8^2_2[ CASP3 ]_c</td>
<td>k_{6r}</td>
</tr>
<tr>
<td>r_{70}</td>
<td>CASP8^2_2[ CASP3 ]_c \rightarrow CASP8^2_2[ CASP3 ]_c</td>
<td>k_{7}</td>
</tr>
<tr>
<td>r_{71}</td>
<td>CASP8^2_2[ Bid ]_c \rightarrow CASP8^2_2[ Bid ]_c</td>
<td>k_{8r}</td>
</tr>
<tr>
<td>r_{72}</td>
<td>CASP8^2_2[ Bid ]_c \rightarrow CASP8^2_2[ Bid ]_c</td>
<td>k_{8r}</td>
</tr>
<tr>
<td>r_{73}</td>
<td>CASP8^2_2[ Bid ]_c \rightarrow CASP8^2_2[ Bid ]_c</td>
<td>k_{7}</td>
</tr>
<tr>
<td>r_{74}</td>
<td>Bid, Bax ]_c \rightarrow Bid, Bax ]_c</td>
<td>k_{9f}</td>
</tr>
<tr>
<td>r_{75}</td>
<td>Bid, Bax ]_c \rightarrow Bid, Bax ]_c</td>
<td>k_{9r}</td>
</tr>
<tr>
<td>r_{76}</td>
<td>Bid, Bax ]_c \rightarrow Bid, Bax ]_c</td>
<td>k_{9f}</td>
</tr>
<tr>
<td>r_{77}</td>
<td>Bid, Bax ]_c \rightarrow Bid, Bax ]_c</td>
<td>k_{9r}</td>
</tr>
<tr>
<td>r_{78}</td>
<td>Bid, Bax ]_c \rightarrow Smac ]_m \rightarrow Smac^* ]_m</td>
<td>k_{10}</td>
</tr>
<tr>
<td>r_{79}</td>
<td>Bid, Bax ]_c \rightarrow Smac^* ]_m \rightarrow Smac^* ]_m</td>
<td>k_{10}</td>
</tr>
<tr>
<td>r_{80}</td>
<td>Smac^<em>, XIAP ]_c \rightarrow Smac^</em> : XIAP ]_c</td>
<td>k_{11f}</td>
</tr>
<tr>
<td>r_{81}</td>
<td>Smac^* : XIAP ]_c \rightarrow Smac^* , XIAP ]_c</td>
<td>k_{11r}</td>
</tr>
</tbody>
</table>
And with the alternative cases and rules of type 96', 96", 97', 97", 98, 99 as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Rule</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{96}$</td>
<td>$Bid \rightarrow Bcl2$</td>
<td>$k_{20f}$</td>
</tr>
<tr>
<td>$r_{97}$</td>
<td>$Bcl2 \rightarrow Bid$</td>
<td>$k_{20r}$</td>
</tr>
<tr>
<td>$r_{96'}$</td>
<td>$tBid \rightarrow Bcl2$</td>
<td>$k_{20f}$</td>
</tr>
<tr>
<td>$r_{97'}$</td>
<td>$Bcl2 \rightarrow tBid$</td>
<td>$k_{20r}$</td>
</tr>
<tr>
<td>$r_{98}$</td>
<td>$tBid \rightarrow Bcl2$</td>
<td>$k_{20f}$</td>
</tr>
<tr>
<td>$r_{99}$</td>
<td>$Bcl2 \rightarrow tBid$</td>
<td>$k_{20r}$</td>
</tr>
</tbody>
</table>

The deterministic kinetic constants (reaction rates) mentioned in the previous table are given in the following; we refer the interested reader to [47] for more details about the rates and references for their estimation. The following tables give the deterministic kinetic rates (reaction rates) used in the description of the reactions;
Table D.1 Kinetics rates for the Fas model

<table>
<thead>
<tr>
<th>$k_{1f}$</th>
<th>$k_{1r}$</th>
<th>$k_{2f}$</th>
<th>$k_{2r}$</th>
<th>$k_{3f}$</th>
<th>$k_{3r}$</th>
<th>$k_{4}$</th>
<th>$k_{5}$</th>
<th>$k_{6f}$</th>
<th>$k_{6r}$</th>
<th>$k_{7}$</th>
<th>$k_{8f}$</th>
<th>$k_{8r}$</th>
<th>$k_{9f}$</th>
<th>$k_{9r}$</th>
<th>$k_{10}$</th>
<th>$k_{11f}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$9.09 \times 10^{-5}$</td>
<td>$1.00 \times 10^{-4}$</td>
<td>$5.00 \times 10^{-1}$</td>
<td>$0.2$</td>
<td>$3.50 \times 10^{-1}$</td>
<td>$0.018$</td>
<td>$0.3$</td>
<td>$0.1$</td>
<td>$1.00 \times 10^{-1}$</td>
<td>$0.06$</td>
<td>$0.1$</td>
<td>$5.00 \times 10^{-1}$</td>
<td>$0.005$</td>
<td>$2.00 \times 10^{-1}$</td>
<td>$0.02$</td>
<td>$1.00 \times 10^{-1}$</td>
<td>$7.00 \times 10^{-1}$</td>
</tr>
</tbody>
</table>

We must add the following aspects to the Fas Membrane System to model HIV-1-related effects:

- $\Sigma' = \{HIVRNA, RT, HIVDNA, HIVLTR, NFAT, CDK9, CyclinT1, PTEFb, HIVLTR: NFAT, Tat, Nef, Vpr, HIVpr, mRNA_{Tat}, mRNA_{Vpr}, mRNA_{Nef}, mRNA_{HIVpr}, HIVLTR: NFAT: Tat, HIVLTR: NFAT: Tat: PTEFb, Vpr: Bcl2, PTPC, Vpr: PTPC\}$.

- $L' = \{n\}$ Besides the other four compartments we have added a nucleus ($n$).

- $\mu = \{s[m \times m \times n \times n \times c]s\}$

- $M_e, M_s, M_c, M_m, M_n$ are the sets of multiplicities (protein multiplicity).

  - $M_e = \{FASL^{1204}\}$

  - $M_s = \{FAS^{6022}\}$

  - $M_c = \{FADD^{10038}, CASP8^{20671}, FLIP^{48779}, CASP3^{12044}, Bid^{15055}, Bax^{50182}, XIAP^{18066}, Apaf^{60221}, ATP^{60221}, CASP9^{12044}\}$

  - $M_m = \{Smac^{60221}, Cyto.c^{60221}, Bcl2^{45166}\}$

- $R'_s, R'_c, R'_m, \text{ and } R'_n$ are the sets of rules which must be unioned with the previous rules (except $R_n$, since it has no counterpart in the Fas model).

  - $R'_s = \{r_{152}\}$
\[ R'_c = \{ r_{100}, r_{101}, r_{103}, r_{104}, r_{129}, r_{130}, r_{132}, r_{133}, r_{134}, r_{135}, r_{136}, r_{137}, r_{141}, \]
\[ r_{142}, r_{143}, r_{145} \} \]
\[ R'_m = \{ r_{144}, r_{145}, r_{146}, r_{147}, r_{149} \} \]
\[ R_n = \{ r_{102}, r_{105}, r_{106}, r_{107}, r_{108}, r_{109}, r_{110}, r_{111}, r_{112}, r_{113}, r_{114}, r_{115}, r_{116}, r_{117}, \]
\[ r_{118}, r_{119}, r_{120}, r_{121}, r_{122}, r_{123}, r_{124}, r_{125}, r_{126}, r_{127}, r_{128}, r_{138}, r_{139}, r_{140}, \]
\[ r_{150}, r_{151}, r_{153} \} \]

<table>
<thead>
<tr>
<th>Label</th>
<th>Rule</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_{100} )</td>
<td>( [HIV_{RNA}, RT]<em>c \rightarrow [HIV</em>{cDNA}, RT]_m )</td>
<td>( k_{21} )</td>
</tr>
<tr>
<td>( r_{101} )</td>
<td>( HIV_{cDNA} [n] \rightarrow [HIV_{cDNA}]_n )</td>
<td>( k_{22} )</td>
</tr>
<tr>
<td>( r_{102} )</td>
<td>( [HIV_{cDNA}]<em>n \rightarrow [HIV</em>{LTR}]_n )</td>
<td>( k_{22} )</td>
</tr>
<tr>
<td>( r_{103} )</td>
<td>( NFAT [n] \rightarrow [NFAT]_n )</td>
<td>( k_{23} )</td>
</tr>
<tr>
<td>( r_{104} )</td>
<td>( CDK9 [n] \rightarrow [CDK9]_n )</td>
<td>( k_{24} )</td>
</tr>
<tr>
<td>( r_{105} )</td>
<td>( [CyclinT1, CDK9]_n \rightarrow [PTEFb]_n )</td>
<td>( k_{25} )</td>
</tr>
<tr>
<td>( r_{106} )</td>
<td>( [NFAT, HIV_{LTR}]<em>n \rightarrow [HIV</em>{LTR} : NFAT]_n )</td>
<td>( k_{26} )</td>
</tr>
<tr>
<td>( r_{107} )</td>
<td>( [HIV_{LTR} : NFAT, Tat]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat]_n )</td>
<td>( k_{27} )</td>
</tr>
<tr>
<td>( r_{108} )</td>
<td>( [HIV_{LTR} : NFAT : Tat, PTEFb]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat : PTEFb]_n )</td>
<td>( k_{28} )</td>
</tr>
<tr>
<td>( r_{109} )</td>
<td>( [HIV_{LTR}]<em>n \rightarrow [HIV</em>{LTR}, mRNA_{Tat}]_n )</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>( r_{110} )</td>
<td>( [HIV_{LTR}]<em>n \rightarrow [HIV</em>{LTR}, mRNA_{Vpr}]_n )</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>( r_{111} )</td>
<td>( [HIV_{LTR}]<em>n \rightarrow [HIV</em>{LTR}, mRNA_{HIVpr}]_n )</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>( r_{112} )</td>
<td>( [HIV_{LTR}]<em>n \rightarrow [HIV</em>{LTR}, mRNA_{Nef}]_n )</td>
<td>( k_{29} )</td>
</tr>
<tr>
<td>( r_{113} )</td>
<td>( [HIV_{LTR} : NFAT]<em>n \rightarrow [HIV</em>{LTR} : NFAT, mRNA_{Tat}]_n )</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>( r_{114} )</td>
<td>( [HIV_{LTR} : NFAT]<em>n \rightarrow [HIV</em>{LTR} : NFAT, mRNA_{Vpr}]_n )</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>( r_{115} )</td>
<td>( [HIV_{LTR} : NFAT]<em>n \rightarrow [HIV</em>{LTR} : NFAT, mRNA_{HIVpr}]_n )</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>( r_{116} )</td>
<td>( [HIV_{LTR} : NFAT]<em>n \rightarrow [HIV</em>{LTR} : NFAT, mRNA_{Nef}]_n )</td>
<td>( k_{30} )</td>
</tr>
<tr>
<td>( r_{117} )</td>
<td>( [HIV_{LTR} : NFAT : Tat]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat, mRNA_{Tat}]_n )</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>( r_{118} )</td>
<td>( [HIV_{LTR} : NFAT : Tat]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat, mRNA_{Vpr}]_n )</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>( r_{119} )</td>
<td>( [HIV_{LTR} : NFAT : Tat]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat, mRNA_{HIVpr}]_n )</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>( r_{120} )</td>
<td>( [HIV_{LTR} : NFAT : Tat]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat, mRNA_{Nef}]_n )</td>
<td>( k_{31} )</td>
</tr>
<tr>
<td>( r_{121} )</td>
<td>( [HIV_{LTR} : NFAT : Tat : PTEFb]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat : PTEFb, mRNA_{Tat}]_n )</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>( r_{122} )</td>
<td>( [HIV_{LTR} : NFAT : Tat : PTEFb]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat : PTEFb, mRNA_{Vpr}]_n )</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>( r_{123} )</td>
<td>( [HIV_{LTR} : NFAT : Tat : PTEFb]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat : PTEFb, mRNA_{HIVpr}]_n )</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>( r_{124} )</td>
<td>( [HIV_{LTR} : NFAT : Tat : PTEFb]<em>n \rightarrow [HIV</em>{LTR} : NFAT : Tat : PTEFb, mRNA_{Nef}]_n )</td>
<td>( k_{32} )</td>
</tr>
<tr>
<td>( r_{125} )</td>
<td>( [mRNA_{Tat}]<em>n \rightarrow mRNA</em>{Tat}[n] )</td>
<td>( k_{33} )</td>
</tr>
</tbody>
</table>
Label | Rule | Rate
--- | --- | ---
$r_{126}$ | $[mRNA_{Nef}]_n \rightarrow mRNA_{Nef}[ ]_n$ | $k_{33}$
$r_{127}$ | $[mRNA_{Vpr}]_n \rightarrow mRNA_{Vpr}[ ]_n$ | $k_{33}$
$r_{128}$ | $[mRNA_{HIV_{pr}}]_n \rightarrow mRNA_{HIV_{pr}}[ ]_n$ | $k_{33}$
$r_{129}$ | $[mRNA_{Tat}]_c \rightarrow [mRNA_{Tat}, Tat]_n$ | $k_{33}$
$r_{130}$ | $[mRNA_{Nef}]_c \rightarrow [mRNA_{Nef}, Nef]_n$ | $k_{33}$
$r_{131}$ | $[mRNA_{Vpr}]_c \rightarrow [mRNA_{Vpr}, Vpr]_n$ | $k_{33}$
$r_{132}$ | $[mRNA_{HIV_{pr}}]_c \rightarrow [mRNA_{HIV_{pr}}, HIV_{pr}]_n$ | $k_{33}$
$r_{133}$ | $[mRNA_{Tat}]_c \rightarrow [ ]_c$ | $k_{33}$
$r_{134}$ | $[mRNA_{Nef}]_c \rightarrow [ ]_c$ | $k_{33}$
$r_{135}$ | $[mRNA_{Vpr}]_c \rightarrow [ ]_c$ | $k_{33}$
$r_{136}$ | $[mRNA_{HIV_{pr}}]_c \rightarrow [ ]_c$ | $k_{33}$
$r_{137}$ | $Tat[ ]_n \rightarrow [Tat]_n$ | $k_{36}$
$r_{138}$ | $[Tat]_n \rightarrow [Tat]_n CASP8$ | $k_{37}$
$r_{139}$ | $[Tat]_n \rightarrow [Tat]_n Bcl2$ | $k_{38}$
$r_{140}$ | $[Tat]_n \rightarrow FASL[Tat]_n$ | $k_{39}$
$r_{141}$ | $[Tat, Bcl2]_c \rightarrow [Tat]_c$ | $k_{40}$
$r_{142}$ | $Vpr[ ]_m \rightarrow Vpr[Bcl2]_m$ | $k_{41}$
$r_{143}$ | $[Vpr, Bax]_c \rightarrow [Vpr]_c$ | $k_{42}$
$r_{144}$ | $[Vpr, Bcl2]_m \rightarrow [Vpr : Bcl2]_m$ | $k_{43}$
$r_{145}$ | $[Vpr : Bcl2]_m \rightarrow [Vpr, Bcl2]_m$ | $k_{44}$
$r_{146}$ | $[Vpr, PTPC]_m \rightarrow [Vpr : PTPC]_m$ | $k_{45}$
$r_{147}$ | $[Vpr : PTPC, Cyto.c]_m \rightarrow Cyto.c[Vpr : PTPC]_m$ | $k_{46}$
$r_{148}$ | $[HIV_{pr}, CASP8]_c \rightarrow [HIV_{pr}, Casp8*]_m$ | $k_{47}$
$r_{149}$ | $HIV_{pr}[Bcl2]_m \rightarrow HIV_{pr}[ ]_m$ | $k_{48}$
$r_{150}$ | $[Nef]_n \rightarrow [Nef, FAS]_m$ | $k_{49}$
$r_{151}$ | $[Nef]_n \rightarrow [Nef, FASL]_m$ | $k_{50}$
$r_{152}$ | $[FASL]_s \rightarrow FASL[ ]_s$ | $k_{51}$
$r_{153}$ | $[Tat]_n \rightarrow Tat[ ]_n$ | $k_{36,r}$
The kinetic rates for the additional HIV-1 rules are given in the table below:

**Table D.2 Additional kinetic rates for HIV-1-infected model**

<table>
<thead>
<tr>
<th>$k_{21}$</th>
<th>0.033456341666667nM$^{-1}s^{-1}$</th>
<th>$k_{36r}$</th>
<th>0.0019s$^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_{22}$</td>
<td>0.000555555555555556s$^{-1}$</td>
<td>$k_{37}$</td>
<td>4.0E - 006s$^{-1}$</td>
</tr>
<tr>
<td>$k_{23}$</td>
<td>100s$^{-1}$</td>
<td>$k_{38}$</td>
<td>2.0E - 006s$^{-1}$</td>
</tr>
<tr>
<td>$k_{24}$</td>
<td>400s$^{-1}$</td>
<td>$k_{39}$</td>
<td>2.0E - 007s$^{-1}$</td>
</tr>
<tr>
<td>$k_{25}$</td>
<td>0.4nM$^{-1}s^{-1}$</td>
<td>$k_{40}$</td>
<td>2.0E - 008nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{26}$</td>
<td>5.0E - 005nM$^{-1}s^{-1}$</td>
<td>$k_{41}$</td>
<td>1.1E - 006s$^{-1}$</td>
</tr>
<tr>
<td>$k_{27}$</td>
<td>0.1nM$^{-1}s^{-1}$</td>
<td>$k_{42}$</td>
<td>2.0E - 008nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{28}$</td>
<td>200nM$^{-1}s^{-1}$</td>
<td>$k_{43}$</td>
<td>2.0E - 008nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{29}$</td>
<td>2.8E - 004s$^{-1}$</td>
<td>$k_{44}$</td>
<td>2.0E - 006s$^{-1}$</td>
</tr>
<tr>
<td>$k_{30}$</td>
<td>2.8E - 003s$^{-1}$</td>
<td>$k_{45}$</td>
<td>2.0E - 006s$^{-1}$</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>0.071s$^{-1}$</td>
<td>$k_{46}$</td>
<td>1.0E - 005nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{32}$</td>
<td>0.71s$^{-1}$</td>
<td>$k_{47}$</td>
<td>6.0E - 012nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{33}$</td>
<td>0.2s$^{-1}$</td>
<td>$k_{48}$</td>
<td>3.0E - 008nM$^{-1}s^{-1}$</td>
</tr>
<tr>
<td>$k_{34}$</td>
<td>0.04s$^{-1}$</td>
<td>$k_{49}$</td>
<td>3.0E - 009s$^{-1}$</td>
</tr>
<tr>
<td>$k_{35}$</td>
<td>0.033s$^{-1}$</td>
<td>$k_{50}$</td>
<td>1.0E - 007s$^{-1}$</td>
</tr>
<tr>
<td>$k_{36f}$</td>
<td>0.002s$^{-1}$</td>
<td>$k_{51}$</td>
<td>2.0E - 006s$^{-1}$</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


[4] G. Baffy, T. Miyashita, J.R. Williamson, and J.C. Reed, "Apoptosis induced by withdrawal of interleukin-3 (IL-3) from an IL-3-dependent hematopoietic cell line is associated with repartitioning of intracellular calcium and is blocked by enforced Bcl-2 oncoprotein production," *Journal of Biological Chemistry*, vol. 268, no. 9, 1993, pp. 6511–6519.


Chemokine Receptor CXCR4 is Induced by Cell Membrane-Associated Human Immunodeficiency Virus Type 1 Envelope Glycoprotein (gp120),” *Virology*, vol. 268, no. 2, 2000, pp. 329–344.


