Modelling and analyses of synthetic biology

Joachim Kirkegaard Friis

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of the thesis is to investigate current means of modelling and analysing chemical reaction systems, in the context of synthetic biology.
Synthetic cells that are proposed to act as electronic gates, called synthetic genetic devices, are simulated under different conditions in order to assess the adequacy of Gillespie's direct method and Oded Maler's proposed model for spatial dynamics.

Both of these models are examined and described in terms of their level of abstraction, i.e. how true-to-nature they are. Experiments are then conducted by utilising a tool proposed and developed for this thesis. The tool itself is designed, such that the models can later be refined and extended. It incorporates a current format for specifying the devices, making it suitable for biochemists to use.

It is concluded that Gillespie's model is in fact sufficient for described the synthetic genetic devices considered in this thesis, under the right circumstances. The motion of the particles, the devices consist of, had a great impact on the simulated dynamic behaviour compared to the expected. This revealed how sensitive the devices are to the parameters of the given simulation.

Keywords: *Synthetic biology; Stochastic simulation; Spatial dynamics; Thermodynamic motion; Automated analysis*

Summary (Danish)

Målet for denne afhandling er at undersøge de nuværende metoder brugt til modellering og analyse af kemiske reaktionssystemer, i forbindelse med syntetisk biologi.
Syntetiske celler, der er ment til at fungere som elektroniske porte, kaldet syntetiske genetiske enheder, simuleres under forskellige betingelser, for at vurdere tilstrækkeligheden af Gillespies direkte metode og Oded Malers foreslåede model for spatial dynamik.

Begge disse modeller er undersøgt og beskrevet i form af deres abstraktionsniveau, dvs. hvor virkelighedstro de er. Eksperimenter er derefter udført ved anvendelse af et værktøj præsenteret og udviklet til denne afhandling. Selve værktøjet er udformet, således at modellerne senere kan raffineres og udvides. Det benytter sig af et aktuelt format til angivelse af enhederne, hvilket gør det velegnet for biokemikere.

Det konkluderes, at Gillespies model er faktisk tilstrækkelig til beskrevet de syntetiske genetiske enheder, der betragtes i denne afhandling, under de rette omstændigheder. Bevægelsen af de partikler enhederne består af, havde en stor indvirkning på det simulerede dynamiske adfærd i forhold til det forventede. Dette afslørede, hvor sårbare enhederne er overfor parametrene for den givne simulation.

Nøgleord: *Syntetisk biologi; Stokastisk simulation; Spatial dynamik; Termodynamisk bevægelse; automatiseret analyse*

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Software Engineering.

The thesis deals with stochastic simulation of synthetic genetic devices, by implementing a modular tool used for experimenting different set ups of such devices.

The thesis consists of both the documentation of the software implemented and the research done in order to refine and experiment on current models of chemical reaction systems.

Lyngby, 02-August-2015

Joachim Kirkegaard Friis

Acknowledgements

I would like express my gratitude to both of my supervisors, Jan Madsen and Michael Reichhardt Hansen, for giving me free rein when initially considering the focus of this thesis. Throughout the work of this thesis, our interesting discussions about the connections between synthetic biology and computer science have kept me highly motivated.

I would also like to thank my family and friends for supporting me during the work of this thesis.

Contents

Chapter 1

Introduction

Synthetic biology is the engineering of biological components that can behave in a predefined way. One can for instance affect the process in which a strand of *deoxyribonucleic acid* (DNA) is split in a cell, such that concentrations of specific compounds change in a restricted and predefined manner. This can then be composed into larger components e.g. biological walkers that can traverse the predefined paths carrying cargo and perform computations [MF13]. This could be done by means of constructing components providing the behaviour of an electronic logical gate, just as we see in computers today. Examples of such cells would be a *negative feedback-* or an *and-gate* device, both keeping a steady signal reflected by certain concentration level of a produced protein within the cell.
In terms of nanorobotics and the relying synthetic components this particular field of study has shown, as of writing this report, potential especially in the field of manufacturing 'empty' cells with the purpose of inserting custom manipulative DNA strings with specific behaviour. But a lot of challenges in terms of creating such in practice are still to be overcome, one being the seemingly random behaviour of chemical reactions within a cell. This motivates research and development of tools for easing the process of *wet lab* experiments for biologists, when they want to test a specific set up.

The purpose of this thesis is to then explore how we can model and analyse such biological systems from the perspective of a computer scientist. The pro-

cess of modelling such systems will provide us with several choices that have to be made in order to narrow down the work of this thesis. The model should describe the behaviour of a given setup of a device consisting of some key components, this is illustrated further in Figure 1.1. The components specified in

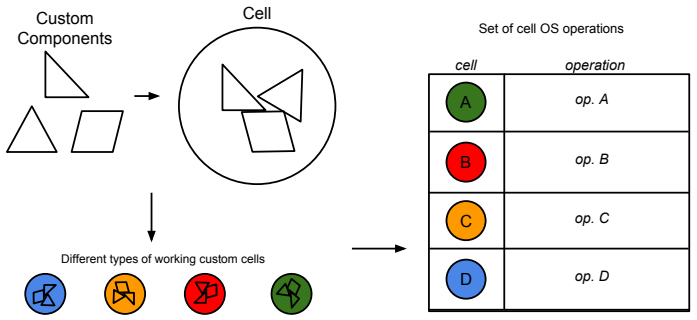


Figure 1.1: An illustration showing a very simplified procedure of inputting some biological components into an "empty" cell in order to gain some predefined behaviour.

the illustration are sequences of DNA strings that should change the behaviour of the given cell. The main purpose of the model is to then apply a means of simulation in order to get an indication on whether the set up of such components will work or not. To give a short analogy, the cell can be seen as an operating system of a computer, where specific mechanics and procedures contribute to the capabilities of the operating system. The same works for a cell, where physical limitations are restricting what set ups work, one being that potential chemical energy stored in a cell is limited and will eventually be exhausted, if not taken into account, after which the given cell will die.

1.1 Core motivation behind synthetic biology

The ability to successfully construct a synthetic cell, that is, a cell containing customs components in order to achieve a specific behaviour as described earlier, would then open doors to potentially revolutionise the field of engineering, comparable to the invention of the first computer, e.g. in terms of energy consumption [LB14]. This will for an engineer spark ideas for countless applications and optimizations of current technology solutions, e.g. rebuilding the computer solely of biological components thus achieving much lower power consumption.

In terms of computation, one of the main ideas is to achieve the behaviour of

general electronics components such as memory storage and logic gates [KC10], which would lead to mimicking the functionalities of a digital computer. Additionally data protection by means of DNA-based encryption is also a possible application, as mentioned in [Wid14]. Another fields of application outside of computation are biosensing, therapeutics, and the production of biofuels [S0:], pharmaceuticals and biomaterials [KC10]. What these have in common, is that they rely on the ability to construct and manipulate the structure and behaviour of the cells as mentioned.

The field of synthetic biology would thus benefit from a collaboration between different domains of engineering, which in this case are the biologists and another given domain, which would in the setting of this thesis be a computer scientist or an electrical engineer. This requires a common ground of terminology and exchange of paradigms that exists in the two domains. This can be quite challenging, since e.g. the term 'model' might have a different meaning to a biologist compared to a computer scientist's. This aspect should serve as a motivation to collaborate in a structured and well-defined manner.

In the context of this, an important note on the terminology used by biologist is the difference between operations done to cells: *in vivo*, that is, the operations are done on living and complete cells, and *in vitro*, that is, operations, such as DNA synthesis, that are done in a controlled environment [Wid14]. And relating back to the analogy in Figure 1.1, the synthetic DNA strand is injected into an empty cell in vitro.

1.2 Current state synthetic biology

Continuous progress is currently made in synthetic biology. It is often proposed to synthesize artificial components in the fields of medicine and biotechnology such as yeast and plants, but more interesting - mammalian cells (cells that mammals (including humans) are made of)[KJKC15]. As stated in [LB14], the cost of DNA *sequencing*, that is, to read the information stored DNA string, is ever decreasing since its discovery and has fallen drastically during recent years due to technological improvements.

Research of synthetic biology is now at a state where different fields of engineering can provide tools and solutions for the different challenges that are now starting to appear. E.g. the practical experiments conducted can be a quite costly process especially in terms of time required. Due to this fact, a tool that could 'filter' out definite faulty set-ups of an experiment, would provide much value to both in an academical and industrial sense. Thus in order to speed up

the engineering of the before discussed biological components, the tool should simulate key parts of the dynamics of a cell, providing valuable insight. This brings the cross communication of different faculties into view - i.e. the terminology of a cell should be adapted into a mathematical model on which well defined and mature frameworks of computer science can be utilised.

1.3 Problem and Goal

The main problem of modelling and simulating chemical reaction systems motivates this thesis. The problems we want to solve in are based on the following questions:

- Which models are adequate for automatised analyses of synthetic genetic devices?
- Are the models proposed by Gillespie and Oded Maler sufficient i.e. true-to-nature when used for simulating synthetic genetic devices?
- And under which conditions do these models work?

The stochastic model used by Gillespie [Gil77] and the *spatial* model proposed by Oded Maler [MHML14] both present an environment for simulating chemical reactions system, but with different arguments on how particles collisions should be modelled. These will then be compared by experimenting with different devices under variable conditions and models describing the environment of the cell itself.
Implementing a framework for comparing the different models, such as deploying a spatial model, is then the main goal when solving this problem. This requires disciplined use of software engineering techniques during the design phase, after which extensions are done to the model in order to evaluate the degree of modularity of the proposed design.

1.4 A framework for synthetic biology

The main purpose of a tool for simulation would be to create an interface linking the qualifications of synthetic biology engineers with the given software system. The most popular solution to do this is the SBML format. This format enables declaration of chemical reaction systems, used in synthetic biology. Below is a snippet of an SBML file:

```
...
  <reaction id="transcription" reversible="false">
    <listOfProducts>
      <speciesReference species="mRNA"/>
    </listOfProducts>
    <listOfModifiers>
      <modifierSpeciesReference species="Plac"/>
    </listOfModifiers>
...
```

As described in [LB14] a framework providing the basis for simulation consists of a parser for SBML files describing a given set up of biological components and their interaction with each other, i.e. one or more chemical reactions. A translator inputs a model resulted from parsing an SBML file. In order to introduce flexibility in terms of running different kinds of simulations, a simulator should then be able to be parametrised. One kind of simulation would be to simulate the behaviour of a synthetic device, running on the premise of described process of central dogma in Chapter 3. Additional simulations could then be done in order to broaden the result space, increasing the reliability and flexibility of the evaluation and analysis that is to be done to the given result. A general analysis that could be done is to evaluate the behaviour described by the simulation with its expected behaviour. The result of the simulation(s) can also then be illustrated in any given way, be it through a graphical user interface, or a simple graph showing the concentration of certain components of a reaction as a function of time. Said framework is illustrated in Figure 1.2.

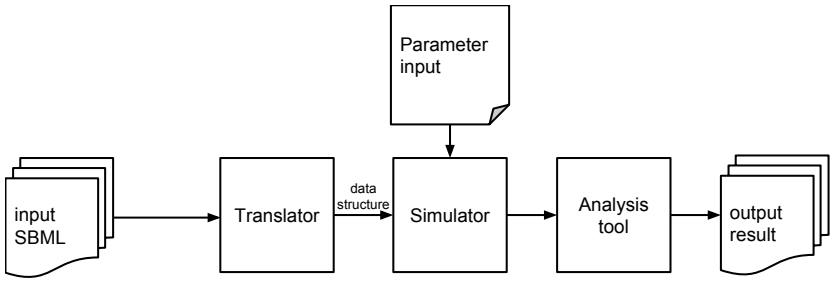


Figure 1.2: Frame work for specifying a synthetic biological reaction, which is simulated with a given data structure and iterated in order to increase result precision.

It should be noted that modularity in this framework is significant due to envisioned extensions and refinements of the illustrated components. Thus keeping

the level of abstraction at a high level and specifying a clear interface between the components in the framework is going to be a key exercise. The reason behind striding for modularity is to gain the ability to easily replace components. E.g. when extensions to the model is to be added, it would be beneficial if it did not require too much code to be rewritten, reused, or restructured. Thus the data structure and the given algorithm for simulation should have close to no relation/knowledge of each other. Choosing existing technologies and frameworks, i.e. using SBML, will serve as starting point in the process of designing and implementing such system, after which the model will be extended in order to improve on the quality of the analyses.

1.5 Structure of the thesis

The structure of this thesis is the following:

- Chapter 2 contains a problem description, detailing the requirements of the tool developed.
- Chapter 3 contains the research done for this thesis.
- Chapter 4 contains a detailed description of the tool implemented, later used for conducting experiments.
- Chapter 5 contains the technical considerations and reflections taken during the implementation, with the focus on parallelised simulations and neighbour search.
- Chapter 6 contains a description of how the tool has been tested, with short examples of such when considering the different components.
- Chapter 7 contains a catalogue of the experiments conducted in order to answer the question stated earlier.
- Chapter 8 contains a conclusion, summarising the tool and experiments, an evaluation of these, and a short section about further improvements and extensions that could be done.
- Appendix A and B contains the SBML files for the devices considered in this thesis.
- Appendix C to S contains the source code of the tool implemented.

Chapter 2

Problem Description

This chapter will outline the fundamental problem that is investigated, by stating the minimum requirements for a framework as proposed in [LB14], with a particular focus on modular components enabling easy model extensions. But first we must discuss the main approach taken during the work of this thesis, which in turn affects the requirements of the tool implemented e.g. if the model and/or analyses should be at focus.

2.1 Approach

As described in Chapter 1, the main purpose of this thesis is to explore the possibilities that arise when we want to model and analyse chemical reaction systems describing synthetic devices. By experimentation we will then test the different models, to see if the their current level of abstraction is adequate. We will do so by testing the devices under different conditions using either model, to see what aspects of spatial dynamics are important to consider.

The main goal is then to model space into the simulations, i.e. taking the particles positions within the cell into account, to see if they are close enough to react. Aside from the software engineering aspects of this thesis, a general

scientific approach in terms of experiments is also motivated in order to test the different stages of the model and compare them with the initial one.

2.2 Problem

Looking back at the illustration in Figure 1.1, the main purpose of constructing synthetic cells is to achieve a defined functionality within the cell e.g. for electronic artificial cells storing information or performing logical operations. A tool for testing a setup of a cell fast and cheaply is then motivated by the alternative of time consuming and costly wet-lab experiments. And through simulation we can gain a deeper understanding of the dynamic behaviour of a given device, by determining the estimated state of the device at each time step of the simulation, which is hard to achieve in an actual wet-lab experiment with current technology.
The overall process in which this tool comes into play is illustrated in Figure 2.1 as an activity diagram in its simplest form.

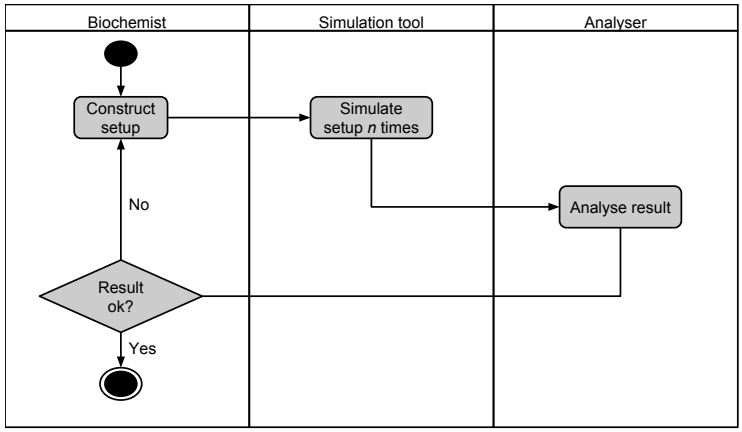


Figure 2.1: An activity diagram with three swimming lanes; one for the actor (biochemist), one for the simulation tool, and one for the given analysis that is conducted on the result of the simulation. The number n is specified by the user and is included in this diagram in order describe the requirement of multiple simulation results.

In Figure 2.1 wee see that the biochemist/user of the system provides the simulation tool with a setup of a cell containing the *species* (particles that take part in the set up), the reaction system describing how the species interact with

each other, and other parameters that describe the environment depending on how refined the working model is. From a given amount of simulations, the analyser then conducts a given analysis - be it a simple graph visualisation and/or extraction of relevant statistics. If the user is then satisfied with the result of the simulation i.e. it confirms his/her hypothesised behaviour, the activity ends, otherwise the user can then alter the set up of the device and conduct a new run of simulations and following analyses.

The main problem is to then construct a tool in which the activity diagram is the main use case with additional features in terms of the underlying model used for simulation and analysis.

2.3 Requirements

The requirements for the tool implemented in this thesis were not formulated at the initial stages of the project, but rather added in an agile fashion. Meaning that initially a rough idea of how the framework looked like, inspired by [LB14], was the fundamental requirement. Any further features in terms of modelling, analysis, and visualisation were later formulated during the project depending on what seemed interesting to research, implement, and experiment with.

The framework for simulating synthetics biological devices/cells will be described in terms of listing the rough requirements - i.e. the functional requirements will be few whilst the non-functional requirements will elaborate on the required quality of the tool. Keeping the functional requirements on a 'rough' level gives more freedom during the design phase in terms of achieving modularity.

The framework described in Chapter 1 consists of the following components: a *compiler*, a *simulator*, and an *analysis tool*. These component are, as mentioned, inspired and build on by the framework proposed in [LB14], and are for that reason required to be in the end product. The requirements are then be described by the basis of the components. As they are illustrated in Figure 1.2, it remains unclear what the exact purpose of each components is. The main purposes and functional and non-functional requirements of these components are as following:

1. **Compiler**: This will simply input an SBML file, supporting the version developed for this thesis, much most likely will change.

   Functional:

(a) It must contain a parser for SBML, that outputs a model reflecting the describe chemical system.

(b) The model must then be compiled into a given data structure.

(c) The parser should be able to parse SBML files of the current version of SBML as of writing this thesis (version 3.1 [HBH+10]).

Non-functional:

(a) The compiler should be maintainable in the sense that the model outputted should be easily modified or replaced.

2. **Simulator**: This will input the data structure reflecting translated SBML file and output the given result of one or more simulations.

Functional:

(a) It must be able to input different kinds of simulation algorithms, including Gillespie's direct method and the spatial algorithm adapted from Oded Maler (detailed in Chapter 3).

(b) It should be able to evaluate different parameters describing the given simulation criteria, such as: an arbitrary number of simulations, a formatting parameter describing which species are of interest[1].

Non-functional:

(a) The simulator could be optimized in terms of performance thus enabling simulations of complex setups while avoiding high run times[2].

3. **Analysis tool:** The exact structure of this component is purposefully kept at an abstract level, since it could be included as a parameter/subcomponent for the simulator in order gain performance. And, as it turned out, is not the main focus of this project.

Functional:

(a) The data from the simulator must be presented through different means of visualisation: graphs, 3D scatter plots, and animations.

(b) It must compute statistics of the data, such as an accumulated average concentration of given species.

(c) It could be able to analyse the data by validating it by comparison of the expected behaviour.

[1] This is not the full list of parameters the simulator could evaluate, but the most essential.

[2] It is of course rather unclear how 'high run times' is quantified, but the purpose of stating this requirement is to maintain disciplined code structure during implementation and to motivate exploration of different techniques used for performance measuring and optimisation in the context of simulating biological systems.

It should be noted that there are no non-functional requirements for the analysis tool, since, as mentioned in the requirements, it will most likely be a part of the simulator and then inherits the requirements of the simulator in terms of performance.

The requirements listed above are few, but the nature of this project, i.e. the element of exploring refinements of how we model synthetic biology, did not allow too specific details of the end product itself. So when the requirements are later compared with the end product, the tests and discussion of the end product will be extending by evaluating the additional implemented features.

Chapter 3

Background

This chapter will start by presenting the biological aspect of this thesis i.e how DNA sequencing and assembly works. This should give the reader the basis for understanding the purpose of simulation and evaluation of the results presented in Chapter 7, where we test the different devices. The devices are then given a detailed description and discussed in terms of their functionality in their given context, what their expected behaviour is, and how it should be compared with a simulation result.
The model presented by Gillespie takes a different approach when modelling particle collisions than the model proposed by Oded Maler. This will be discussed by comparing the models in terms of how they describe a chemical reaction systems in relation the devices and the cells they are reside in.

3.1 Manipulation of DNA

The purpose of this section is to give the reader sufficient information, to understand and evaluate the work done in this thesis. The following description is thus not meant to be detailed in any sense, but to give a computer scientist, without any preliminary knowledge about the topic at hand, a rough idea about the mechanics of DNA replication in a cell etc. If needed, a much more detailed

description of such can be found in [LB14].

DNA is the building blocks for any kind of *mammalian*, *bacterial* or *viral* cells, that is, e.g. the building blocks of life as we know it. It contains deep information about how the given body should be build - from very basic functionalities to refined characteristics, that makes every living being unique. DNA is a double helix storing information by allocating bases (*(A) adenine, (G) guenine, (T) thymine, and (C) cytosine*) in a restricted manner. In Figure 3.1 a small part of an example DNA is shown.
The main purpose the double-helix structure of DNA is for greater robustness, i.e. if one helix is damaged the other can be utilised instead. This is achieved by the bound created between the bases, these bounds are restricted such that adenine can only bind to thymine and genuine and only binds with cytosine.

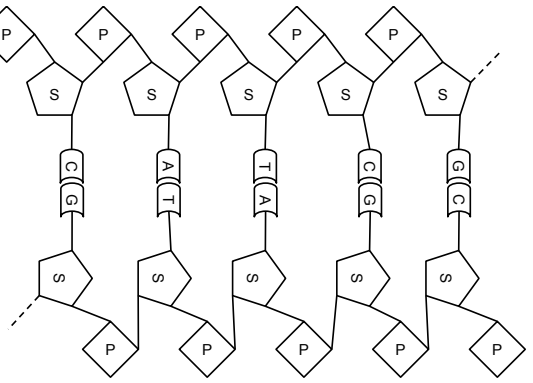


Figure 3.1: An example illustrating the structure of the DNA double helix in which the base pairs are bound to each other following the binding rules. Each pair is connected to a sugar, which is then connected to phosphate connecting the whole structure.

In order to read this information, a DNA string is split in half such that the bases are exposed. This splitting happens when DNA needs to be replicated in order to create new cells. In this process other important *macro-molecules/nucleotides*[1] are to be mentioned; the proteins and the *ribonucleic acid* (RNA) involved in the process. It should be noted that RNA exists in different forms, each having its own purpose; mRNA (messenger), sRNA (small), and tRNA (transfer), though their purpose is out the scope of this thesis.

Genes are small stretches of a given DNA strand. They utilise the information stored in the DNA to produce a gene product. This product is either an

[1] consists of molecules of relative smaller molecular mass

RNA or a protein, where the protein is in our particular interest as it is in [LB14].
The process known as the *central dogma of molecular biology*, where the information stored in DNA is read, is illustrated in Figure 3.2. The protein structure is controlled by components better known as *regulatory segments* of the given DNA strand. These are the *promoters, ribosome binding site (RBS), protein coding sequence (PCS), and the terminator [LB14].* These components affect the process described in Figure 3.2.

- The process in which the mRNA is synthesized is called *transcription*. Initially a DNA strand is split in two, an enzyme RNA polymerase sits on one of the strand and produces a mRNA that is matched by the exposed bases. It should be noted that many polymerases can sit on a given strand, resulting in concurrent mRNA production. The production stops when the polymerase meets a terminator.
- As then seen on Figure 3.2 the mRNA is *translated* into specific amino-acids that are the components of a protein. An important aspect of this the information space introduced by the different possible type of amino-acids i.e. 20. Although there exists 64 different *codons*[2].

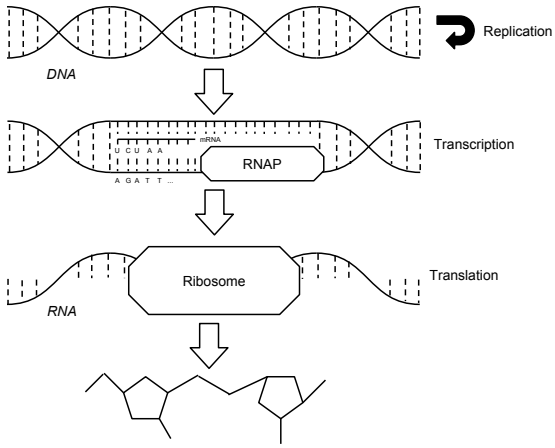


Figure 3.2: The process known as the central dogma, where DNA is replicated such that genes can be translated.

A lot of aspects in terms of transporting the RNA's in transcription and translation cause random fluctuations of how much protein is generated in the given

[2]A sequence of three bases e.g. A-G-G

process. The components described can also simply decay in the process, which turns out to have an important impact of gene expressions. This random element is crucial to the understanding of the mechanics of DNA sequencing and is the general foundation behind the analyses and modelling done in this thesis. An example will later be given in this chapter, outlining the relation between the stochastic and desired behaviour of said mechanics.

An important aspect of the genes central to production of proteins is the possible interaction between the promoter and the produced proteins, this is called *gene regulation*. When regulation occurs the amount of protein is regulated, since the promoter is 'turned on or off' respectively caused by *inducing* or *repressing* proteins. This effect causes a *steady-state* of proteins in the cell. As concluded by [LB14], this behaviour can be compared with the on- and off state of an electric transistor.

Sequencing, synthesising, and assembly

When talking about synthesising DNA, we do not only talk about creating custom DNA but also to combine parts of different strands to create a new one. But before we can synthesise, we must be able to sequence DNA on our own, which is to obtain information about the base pairs in the given DNA strand. This can be achieved in numerous ways, but [LB14] describes the *Sanger sequencing*.
In short it emulates the transcription phase discussed earlier, by splitting a given DNA strand in two - a *template* and a *complementary*. The template is then mixed with a polymerase in four different separate containers, after which a mixture of nucleotides and a PCR[3] are put in as well. The strand then repairs itself in a unique manner different to each container. From this the sequence of base pairs in the given DNA strand can be determined. Sequencing and assembly of DNA strand then allows the creation of artificial DNA strand - i.e synthetic DNA strand through DNA synthesis. In this process strands of few base pairs are coupled together forming a larger strand. This enables insertion of such strand into an 'empty' cell, afterwhich a given dynamic behaviour is expected given the process of the central dogma described earlier.
This process is rather costly in terms of time needed for creating a specific set up of a custom DNA strand. Motivating the tool implemented in this thesis.

[3] a technology used to generate a high number of copies of the DNA strand

3.2 Engineering synthetic genetic devices

In order to simulate a biological system, more precisely - its set of chemical reactions, important choices must be made in terms of abstractions from the real world. In theory, one could simulate a cohesive true-to-nature model representing reality. But many indeterminable variables cause unreliable behaviour in 'chaotic' chemical reactions systems, i.e. a system sensitive to its initial conditions. Such behaviour is in probabilistic sense called stochastic, meaning that a state of a system is randomly determined. Further meaning, that we cannot precisely predict the outcome of a given reaction, but we may apply statistical analyses in order to conclude something from a simulation.

What can we simulate?

The general purpose of simulation in this context is to estimate the behaviour of a given setup of a device, on basis of the process described earlier. Such device is specified by the user of the system, on which the simulation is done and a set of analyses can then be applied. What we can simulate is then directly related to the model and how true-to-nature it is. An example of a device is a simple negative-feedback device, as proposed in [LB14], can be seen in Figure 3.3. This device describes a specific gene where the produced protein provides a negative feedback loop on the promoter itself, restricting the concentration of the protein, such that it reaches a point of repression at a given time, i.e. steady-state. The notation used for the device is an adaptation of the *Synthetic Biology Open Language* (SBOL) which is notated below the device in Figure 3.3.

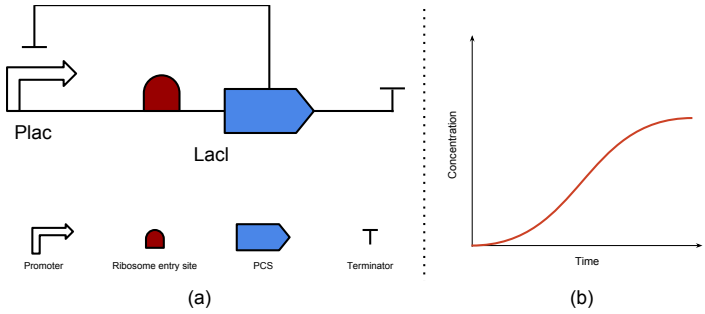


Figure 3.3: (a) The negative feedback device described in a special notation used in [LB14]. (b) Its expected behaviour of repression of the LacI protein at some point in time, since the chance of LacI reacting with the promoter is proportional to the amount of LacI.

It should be noted that the graph to the right shows the average behaviour of the device, meaning that the result of a single simulation would show 'spikes' in concentration of LacI. Upon reaching the steady-state the concentration of LacI is expected to stay within a certain interval. This interval is then the key factor of evaluation, later done in the experiments in Chapter 7.

As stated in [LB14], the model presented by Gillesipe contains abstractions from the real world. One example of how this model can be extended, is to introduce the dynamics of mass action systems. That is, a system with *species* reacting with each other only upon actual contact, not only based on probable estimation. Meaning that a *reaction rule* will only occur if the related species are in fact close enough to each other.

Another suitable example to introduce, would be a device exerting the same behaviour as a logical AND-gate. The SBOL representation of this device can be seen in Figure 3.4.

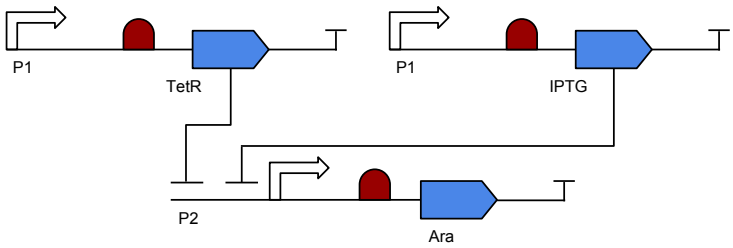


Figure 3.4: A device that behaves the same way as a logical AND-gate, given that the promoter P2 is induced by *both* of the proteins produced above.

Here we see two inducing proteins interacting with the promoter (P2), thus activating/turning it on. The expected behaviour of this device can be compared with that of the negative feedback device - that once the promoter is activated, we should see steady-state of the Ara protein due to the limit of induction of P2. The purpose of chosen this example will become much clearer, once we introduce the extended model - taking the particles position into account i.e. given the fact that if the two inducers are not near each other they should not activate the promoter, resulting in a different behaviour.

3.3 Quantitative and stochastic simulation

In this section the different means of simulation will be introduced and compared. Introducing the simplest form of modeling chemical reaction systems by deterministic analysis, will be introduced to show how a "too" simplified approach would lead to unrealistic results. This motivates an extended model, by introducing stochastic elements and spatial dynamics. Once the spatial model, proposed by Oded Maler, has been introduced, research into how particles move in a cell has to be done to further refine the model to be as true-to-nature as possible.

A key challenge when simulating the models of devices, introduced earlier, is to find suitable initial parameters: such as reaction rate, size of the different species relative to the cell, and the viscosity of *cytoplasm* (the majority fluid in a cell) etc.. The is the main limitation when simulating biological system is finding reaction rates, which is very difficult to measure by experimentation.

Deterministic and stochastic methods

There are two distinctly different ways to simulate a chemical reaction system composing of a set of *species* and the defined rules of reactions. One is the deterministic and continuous technique, such as solving a set of *ordinary differential equations* (ODEs) describing the *laws of mass action* [LB14]. This model does not reflect reality, since chemical reaction system describe a stochastic system i.e. the model should over-approximate in terms of fluctuation in rates of which species react with each other. By over-approximation, we mean that the resulting population sizes of a given species, when simulation of the same device is repeated, should not evaluate to a specific amount but rather an interval of which it might be in. The fluctuations between repeated simulations should also provide deeper understanding of the mechanics of the devices.

A popular stochastic simulation algorithm, proposed by Gillespie [Gil77], is the *direct method*. A procedure of running the direct method is then to obtain a set of resulting simulations and average them in order to get a sense of behaviour of the given system.

What do the ODEs of a reaction describe?

Before describing further, we should formalise a chemical system. As defined [Gil77], such system consists of a set S of n species, where a set R of m reactions defines the reactions between the species in S. A factor that defines the rate of which a reaction happens is the rate function $\lambda_{1,..,m}$. When a reaction occurs state change vectors $v_{1,..,m}$ describe the change of each species. The *Predator-Prey* example is often used to illustrate such system, in which population growth and decay of predators and prey in a forest changes over time. As described in [LB14], the ODE's for this system are as following:

$$\frac{d[Prey]}{dt} = k_1[Prey] - k_2[Prey][Predator] \tag{3.1}$$

$$\frac{d[Predator]}{dt} = k_2[Prey][Predator] - k_3[Predator] \tag{3.2}$$

In equation 3.1 and 3.2 we see that the rates of each population is dependent on each other. The equations are formalised through the *law of mass action*, where each reaction in the form of $X_0 + .. + X_i \rightarrow Y$ has a rate function defined λ_μ defined as:

$$\lambda_\mu = k_\mu \prod_{S_i \in \cdot \mu} X_i \tag{3.3}$$

Given equation 3.3 we see that the reaction rate of a given reaction is proportional to the amount of each population/species included in the reactions, hence the intuition behind having an increase amount of particles in an isolated system with spatial boundaries, the chance of them interacting increases. This is the basic principle behind this model, which is the point of investigation of this thesis:

Is it enough to model particle interaction through deterministic calculations, statistical estimation through stochastic simulation, or through real-time simulation of the particles exact movement and position?

Given this property of reaction rates provided by the law of mass action, we see that it would reflect the expected behaviour of the devices proposed earlier in Figure 3.3, as the given amount of produced proteins increases. We see the same behaviour in the Predator-Prey system [LB14], in which the rate of prey reproduction is proportional to the amount of prey present. Predators reproduce by consuming prey and is also proportional to the amount of predators present etc.

The model described by ODEs leaves out any fluctuations that could happen in

this system, if it was set in the real world, where we increase the amount of unknown variables i.e. add a stochastic element. One could for instance ask: what if some of the prey got smarter, and would not always be caught when hunted by the predators? How would this be modeled into a deterministic system?

Stochastic analysis using Gillespie's direct method

Adding a stochastic element to a model is done when we cannot definitely determine a factor within a system, be it by empirical knowledge or logical argumentation. By adding stochastic behaviour to a system, such as the chemical reaction system described earlier, we would get different results of behaviour at each simulation. This will provide us with deeper knowledge about different possible states of which the system can be in. Furthermore, evaluation of such system can be supported by utilising statistical methods such as comparing different setups or models, to see if they reflect the same behaviour within a certain degree of confidence.

There are many possible ways of achieving this, be it by statistical model checking or statistical evaluation of simulation results. In this case, we will focus on extending the model proposed by [LB14] as described in Chapter 2 and compare different versions of it by means of statistical evaluation of simulation results.

A stochastic system as proposed by [Gil77], describes a system in which particles move around in space with a statistical estimation of collision. This extends the deterministic model, by not always 'allowing' a reaction R_μ to happen, if it by the current state of the system has a too low probability compared to the other reactions in R_m.

Other means of describing reaction system do exists. One would be to, as proposed in [BFR08], construct high-order conditional multiset rewriting, taking a more generic approach on how to compute and extended on current models. But as mentioned earlier, Gillespie's direct method is a broadly used model thus motivating the investigation of its spatial abstractions later discussed.

The stochastic petri net

In terms of data structures, there are different ways of describing a chemical reaction system, or in a more general sense - population systems. One could for instance choose to just keep the entire population/particles of the system in one dictionary, uniquely identifying each particle, providing fast search queries.

But a dictionary is not suitable for a dynamic environment, when we want to dynamically 'move' them around or keep tracks of population sizes. This fact illustrates the importance of choosing a suitable data structure when we start proposing a model.

Petri nets are powerful when modelling biological systems, as they model discrete continuous systems. They provide a nice graphical notation, which can be extended, leading to a wide range of applications. There are many different classes of Petri nets, but the one we are interested in, is the Stochastic Petri net (SPN). The SPN is used to describe a quantitative time-dependent system [MAB11], in which the before mentioned stochastic behaviour can be incorporated. The general Pteri net has the following formal definition [MAB11]:

Definition 1 *(Standard Petri net) A standard Petri net is a quadruple* $N = (P, T, f, m_0)$*, where:*

- P*,* T *are finite, non-empty, disjoint sets.* P *is the set of places.* T *is the set of transitions.*
- $f : ((P \times T) \cup (T \times P)) \to \mathbb{N}_0$ *defines the set of directed arcs, weighted by non-negative integer values.*
- $m_0 : P \to \mathbb{N}_0$ *gives the initial marking.*

Let us consider to following example of a Petri net with a modification, as seen in Figure 3.5:

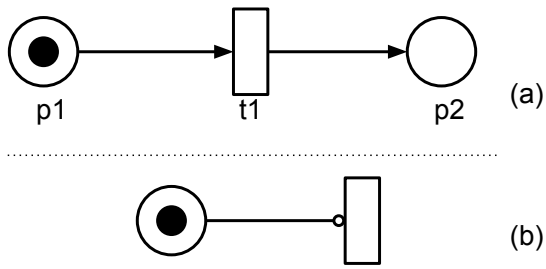


Figure 3.5: (a) A simple example of a Petri net. (b) The modifier arc graphical notation.

Here we have a Petri net consisting of two places, two arcs, one transition, and one token. This Petri could e.g. describe the chemical reaction of $A \rightarrow B$, where A is the reactant consumed in order to create the product B. The state of this reaction is then described by the initial marking m_0, which in case is denoted by the single token residing the first place p_1. If the chemical reaction then occurs, the transition t_1 is then "fired", after which the token is consumed and a new one is created in p_2. An important note on firing should be taken. The general rule for firing a transition can be described as following:

When a transition t is fired, it is first checked if it is *enabled*, that is, if all places p_{in} of all incoming arcs have atleast one token. If so, one token is consumed from each place in p_{in}. All places of outgoing arcs p_{out} then gets a new token added.
Below the Petri net example in Figure 3.5, an extension called the 'modifier are' has been proposed by [LB14] and is also utilised in this project. The modifier arc alters the firing rule, by not consuming tokens in p_{in} when firing its transition. This enables simple modelling of reactions in the form of $A + B \rightarrow A + C$, where e.g. A can be seen as the promoter in device 3.3 - the promoter is of course not consumed when it produces mRNA.

Firing a token happens *instantaneously* and does not consume any time. Meaning that, when firing a token we are simply talking about transitioning from one marking to another in a discrete manner.
The basic Petri presented can then be extending by adding stochastic functionality in T, meaning that transitions also have chance of not being fired upon being selected, even though they are enabled, when transitioning to a new marking. This is done by adding a probability density function, describing the chance of firing a transition proportional to the time elapsed since enabled. This function is defined as:

$$f_{T_\mu}(\tau) = \lambda_\mu \cdot e^{-\lambda_\mu \cdot \tau} \tag{3.4}$$

Where T is an exponentially distributed random variable ranging from $[0, \infty]$. Wee see, that the law of mass action is used in order to model the increasing likelihood of reaction proportional to the population size of the given species. This function is then adapted into the Petri net, defining the SPN.

Looking back at the negative feedback device illustrated earlier, we can now construct an example when it is transformed into an SPN. Before doing so, we must first declare the reaction system in terms of stating the reaction set R:

$$\text{Plac} \xrightarrow{c_1} \text{Plac} + \text{mRNA}$$
$$\text{Plac} + \text{LacI} \xrightarrow{c_2} \text{PlacLacI}$$
$$\text{PlacLacI} \xrightarrow{c_3} \text{Plac} + \text{LacI}$$
$$\text{PlacLacI} \xrightarrow{c_4} \text{PlacLacI} + \text{mRNA}$$
$$\text{mRNA} \xrightarrow{c_5} \text{mRNA} + \text{LacI}$$
$$\text{mRNA} \xrightarrow{c_6} \emptyset$$
$$\text{LacI} \xrightarrow{c_7} \emptyset$$

The last two reactions denote the decay of mRNA and the produced LacI protein. c_{R_n} denotes the reaction rate, which is part of the input when simulating, which will be described in further detail in Equation 3.5. The resulting SPN for this system is then illustrated in Figure 3.6.

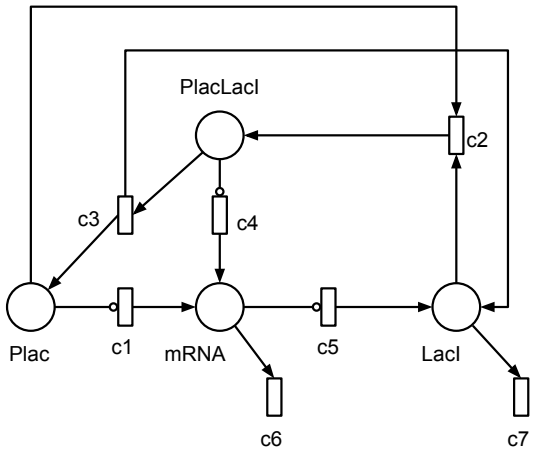


Figure 3.6: The reflected SPN of the negative feedback device in Figure 3.3

Simulation algorithm

Gillespie's direct method is one method of analysing and simulating a chemical reaction systems. The simulation algorithm gives a random result of the given setup of a device, which depends highly on the inputted parameters. The overall flow of the algorithm is as shown in Figure 3.7. For a more detailed description of this algorithm, please refer to [Gil77] or [LB14]. But the the fundamentals

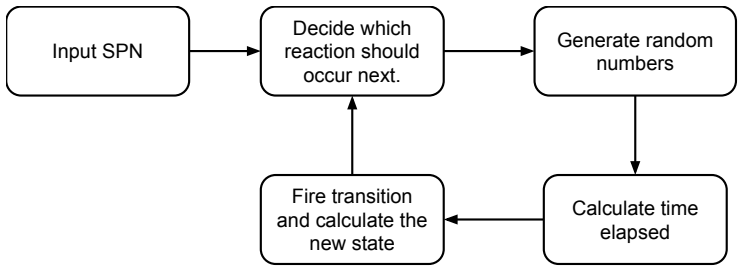


Figure 3.7: A flow diagram showing the general process of Gillespie direct method. The algorithm terminates when sufficient time iterations (loops of this diagram) have been done.

are as following:
The variables a_j and a_0 are respectively the *propensity function* of each reaction/transition, and the sum of all propensity functions. The propensity of each reaction is, again, adapted from the law of mass action. Thus it is defined as:

$$a_\mu = c_\mu h(\mu) \tag{3.5}$$

Where c_μ is the rate constant k_μ, and $h(\mu)$ is the product of all quantities of each species, as seen in equation 3.3. It is important to note that c_μ is a statistical estimation of particle collisions, which leads to the coming discussion about the level of abstraction that this model takes in terms of describing how and when particles collide in a given environment.

The time between each iteration of the simulation then depends on the generated variables, which is defined by: $\tau = (\frac{1}{a_0})ln(\frac{1}{r_1})$, where r_1 is one of the randomly generated numbers taken from a uniform distribution. From this, we can see that when a_0 increases, i.e. as the population grows, the time steps decrease - illustrating that a higher amount of particles increases the amount of collisions happening. This can also be formalised as - when the environment gets progressively more "well-stired"/uniformly distributed the greater the probability of reactions occuring is. To combine this, we can now describe the algorithm by the following pseudo-code:

Data: Stochastic Petri Net describing a chemical reaction system.
Result: A set of states of the system of each time step.
set t = 0 and n = 0;
while $n < max$ **do**
  Calculate a_j and a_0;
  Generate two random numbers r_1 and r_2;
  Take $\tau = (\frac{1}{a_0})ln(\frac{1}{r_1})$;
  Take μ that is smallest of a_j such that $a_\mu > r_2 a_0$;
  Put $t = t + \tau$;
  Fire transition of a_μ and calculate next state;
  Put $n = n + 1$;
end

Algorithm 1: Gillespie's direct method

Level of abstraction

The argument behind taking a statistical approach when simulating particle interaction is best described by illustration in Figure 3.8. Two molecules/particles S_1 and S_2 are spheres traveling in a closed volume. They respectively have *reactionradius* r_1 and r_2. They then collide if their relative distance $d < r_1 + r_2$. Their velosities can then describe a *reaction volume* at each time step, and a statistical estimation can be done in terms of the reaction rate of where the particles are included.

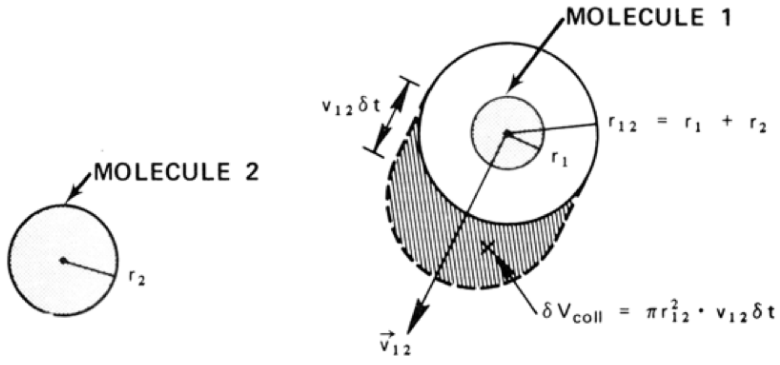


Figure 3.8: Illustration from [Gil77] showing the 'reaction volume' of a particle relative to another.

This describes which level of abstraction this model is at. A question then arises:

Does this estimate in fact describe collisions sufficiently?

We do know that Gillespie's model takes Brownian motion into account when estimating collision, but as it turns out, the movement of particles is highly dependable on the given thermodynamic model used for describing the motion. Determining the motion of particles in a cell is a point of research in itself, in which many parameters are discussed in terms of their affect on Brownian motion. In Gillespie's model it is stated that "Since the system is in thermal equilibrium, the molecules will at all times be distributed randomly and uniformly throughout the containing volume V."[Gil77]. This is a crucial assumption, in which the exact velocities vector are now out of context.

This motivates the next iteration of model extension, i.e. investigation of dynamics of mass actions system. In which it is interesting to see if distribution of particles affect the reaction rates in R.

3.4 Dynamics of mass action systems

Another mathematical model describing a population system, in which species react be means of reactions as in the chemical reactions system described earlier, is proposed by Oded Maler in [MHML14]. This model is more generalised towards *mass action systems*, that is, any population system, be it a social network or other, in which the law of mass action describes a polynomial relation between reaction rates and population sizes, as we seen before in Equation 3.3.

In the paper [MHML14], in which this model is proposed, different stages of the model is investigated, starting from a standard model corresponding to the ODEs described earlier. They then refine the model by adding stochastic behaviour, much like the model proposed in [Gil77], under the same assumption - that particles are "well-stirred" i.e. uniformly distributed, thus not including space. Their last iteration of the model then takes space into account, by keeping track of the particles positions as they move at each discrete time step. This model is referred to as taking *individual spatial dynamics* into account.
A tool **Populus** is then presented, from which they conduct a few experiments in order to compare the different models. They can then evaluate on the hypothesis that abstracting away from spacial dynamics, has an effect on the stochastic behaviour i.e. the behaviour of reaction systems.

Probabilistic automaton

The species of the system and the reactions are described similarly to the SPN, but as a Probabilistic Automaton (PA). The PA will not be described in the same level of detail as the SPN, but it is still interesting to compare the two, and see if they model the same system.

In short, the PA describes a transition system of a set Q of n species. An example of such can be seen in Figure 3.9.

δ	$\perp$			q_1			q_2			q_3		
q_1	0.9	0.1	0.0	1.0	0.0	0.0	0.7	0.2	0.1	0.7	0.0	0.3
q_2	0.1	0.8	0.1	0.0	0.6	0.4	0.0	1.0	0.0	0.1	0.9	0.0
q_3	0.0	0.0	1.0	0.7	0.0	0.3	0.3	0.4	0.3	0.0	0.0	1.0

Figure 3.9: An example of a PA, where the probability functions of for all combinations of reactions are shown in a table.[MHML14]

A transition $\delta(q_1, q_2, q_3)$, referred to as a *binary rule*, then refers to the reaction $q_1 + q_2 \to q_3$, i.e. when q_1 collides with a particle of type q_2, it then produces a particle of type q_3. $\perp$ denotes a spontaneous reaction, e.g. a reaction $q_1 \to q_3$, which transition is referred to as a *solitary rule*. From the transitions table we then see that e.g. the probability of the reaction $q_2 \to q_1$ occurring is 0.1.

Comparing this with the SPN, we see that the entire possible state space of the system is covered, which is not the case in SPN. The SPN only contains the necessary transitions, where the PA's generic nature results in a larger set of transitions. It would still be possible to describe the device through a PA, but the SPN also provides stronger graphical fidelity, easing the process when a given device is modeled by a biologist. Let us consider a much more complex device, compared to the ones described earlier, consisting of a larger set of reactions. The PA would then grow exponentially whilst the SPN grows proportionally to the amount of reactions. So, even though the PA provides a good foundation for a lot of applications, it would in practice be rather cumbersome to model a complex chemical reaction system.

So if we wanted to extend our model, such that it takes space into account. It is motivated to extend the SPN, by first considering tokens at individual particles. So when firing a transition with two or more incoming places, we simply find a pair of particle/tokens colliding. But we should also consider how this should be simulated.

Algorithm for individual spatial dynamics

The algorithm for simulating a time step in the system proposed in [MHML14] is listed in Algorithm 2. A list of particles described by their type and coordinate in the two-dimensional place is the input. The particles are then initially moved. Then, for each particle, its neighbours are computed and the related rules in the PA are applied.

It is important to note, that when a particle has more than one neighbour, all the relating binary rules are applied after which one of the outcomes are randomly chosen. The reason for not just initially chosing one neigbour at random and then apply that one rule, seems unclear.
When comparing this algorithm with the direct method described in 1, we should note that this one works in discrete time. This differs from the propensity functions in Equation 3.4, that describes an increasing probability as a function of time enabled.
But the behaviour described by kinetic law of mass action is still maintained. Lets consider we have one type of particles A, the chance of them being included

Data: A List L of particles and states including planar coordinates.
Result: A List L' representing the next micro-state.
L:=$\emptyset$;
L':=moveParticles in L;
foreach *particle* p *in* L' **do**
  N:=findNeighbours for p in L';
  if $N = \emptyset$ **then**
    q:=apply solitary rule for p;
    L':= insert q into L' ;
  end
  else
    M:=$\emptyset$;
    foreach n *in* N **do**
      q:=apply binary rule for p and n;
      M:= insert q into M ;
    end
    L':= insert random q from M into L' ;
  end
end

Algorithm 2: Oded Maler's algorithm for individual spatial dynamics

in a reaction in Algorithm 2 increases proportionally to the size of its population, and when we apply the rules we will maintain the stochastic behaviour. But the abstraction of discrete time will not allow us to generate appropriate results, which motivates a 'mix' of the two algorithms, based on the direct method in which we find and choose neighbouring particles for reaction following the this method.

Level of abstraction

One interesting abstraction taken by the individual spatial model, is the addition of geometric limitation provided by the description of *periodic boundary condition*. This is described by a rectangular boundary that causes particles moving outside of the rectangle to re-appear on the opposite side. This effect is illustrated in Figure 3.10.

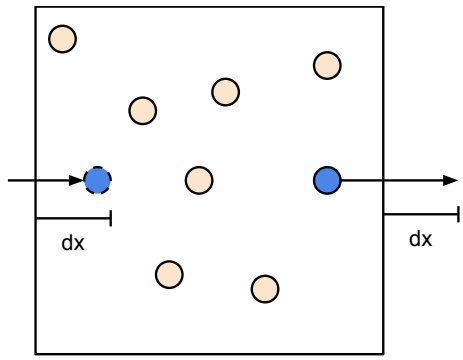


Figure 3.10: Illustration of the *periodic boundary condition*, where a particle (blue) reaches the limit of the volume considered in the given simulation. The particle then 'teleports' to the opposite side, by a displacement of the remainder of the movement vector.

This is a common technique used when simulating any kind of spatial system. It describes an infinite surface on which the particles traverse. In geometric terms, it is a *torus* i.e. 'donut' shape. Whether this is suitable for simulating a real biological cell, seems rather unclear. One could assume that this abstraction would suffice at an increasing density of particles, as we get closer to a uniform distribution. But in the opposite situation, a poison distribution of particle would result in rather unrealistic behaviour once a cluster of particles reaches a boundary. These two situations are illustrated in Figure 3.11.

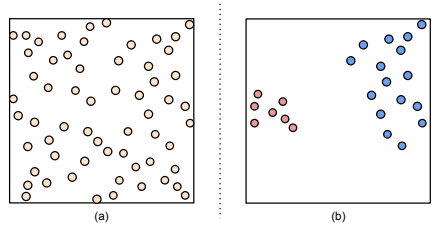


Figure 3.11: Two examples where the periodic boundary condition has close to no effect (a) on the behaviour, and another (b) having a great impact on the behaviour. By 'behaviour', we mean how close to reality the results are of a simulation when using the periodic boundary.

Here we see that in the case of poison distribution, a cluster of particles might

appear. If this cluster reaches the boundary, some of its particles (red) will be out of reach for reacting with the rest (blue). Given the description of the central dogma given in Figure 3.2, intuitively we might think the device as a cluster of particles in which the said mechanics happen. For this reason, the boundary condition will not suffice as means of describing the membrane of the cell. An implementation of a physical wall, where particles simply 'bounce' back upon impact, is then needed (described in detail in Chapter 4).

Another limitation of this model is the motion of the particles, which are described as random displacement within a circle. How this relates to actual movement of particle in a cell, is at this stage unclear, which motivates the next iteration in this chapter.

The experiment results presented in [MHML14], obtained by using the Populus tool, indicate that by initially placing particles closer to each other result in higher reaction rate, compared to that of a uniform distribution. How reliable their results are is at question, since they do not provide any structured presentation of their experiments with the parameters they used. But the inverse of this observation would be interesting to test - i.e, *would an increased distance between particles result in lower reaction rates?*. Later addressed in Chapter 7.

3.5 Thermodynamic motion

How well the before mentioned models are suitable for simulating genetic devices in a synthetic cell, relies much on the parameters describing how the different species move and interact within the cell. One could argue, that the components of the devices described earlier, do in fact *not* move like gaseous particles, i.e. not resulting in a uniform distribution. This could be caused by a possible higher viscosity of the cytoplasm they travel through, and the more complex structures of the components compared to simple molecules, both causing less rapid movement.
This section will discuss what possible extensions we can do to the model, in terms of motion of the particles.

Brownian motion

Brownian motion is in itself a field of research, but for the purposes of this thesis, we will take a short look at its formal definition in order to access its properties. This will provide us with an understanding of said motion, such that we can later implement and experiment the devices in terms of motion patterns.

Brownian motion describes a random walk on d-dimensional space. A particle is displaced in a random fashion by a vector v, describing possible particle-to-particle- or external force interaction. As formalised in [MP11], such random walk should have the following properties:

- Each iteration of the walk should be independent with its previous iterations, called *independent increments*. E.g. the direction of the displacement does not rely on the previous displacement.
- Each iteration of the walk should be have *stationary increments*. E.g. the 'size' of the displacement does not rely on the previous displacement.
- The entire random walk has 'most surely' a continuous path. Meaning, a particle does not 'vibrate' within an area, but moves unrestricted throughout the d-dimensional space.
- The displacement vector v is normally distributed (with additional properties that are out of scope of this thesis).

From a physical point on view, we can see that Brownian motion does not reflect inertia of particles or sense of direction. This will simplify the model by excluding complex physical variables, by providing a simple mathematical notion of Brownian motion instead.

In terms of the experiments done in Chapter 7, normally distribution vectors will be compared with uniformly distributed vector of displacement. Given the complex nature of the biological devices, it is hard to predict if this has an effect on their behaviour, which motivates the experiment itself for the purpose for simply exploring the given model implemented.

Thermodynamic models of motion

The systems described in this chapter can be defined as thermal dynamic systems. Such systems have defined properties such as the state of equilibrium.
Earlier we mentioned that [Gil77] assumes thermal equilibrium, resulting in a uniform distribution of particles when particles are also assumed to behave as gasses. Thermal equilibrium is defined as stable state of system that is not affected by external forces. But as the following description of Brownian motion provides, we know that it includes external forces on the particles of the system. And as it turns out, most thermodynamic system are never in reality at thermal equilibrium.
This motivates a model that is able to include some description of Brownian motion depending on a given thermodynamic model.

The most basic model that determines the velocity of particles within a thermodynamic system is derived from kinetic theory, in which the energy of a particle is defined as following [Kit71]:

$$E_k = \frac{1}{2}mv^2 = \frac{3}{2}kT \tag{3.6}$$

where m is the mass of the particle, v is its velocity, k is the Boltzmann constant (used instead of the ideal gas constant, when we are considering exactly particle amount rather that substances), and T is the temperature in Kelvin. If we wanted to use this formula for describing the velocity of the particles of the device, it clearly provides some limitations. Since it is used for modeling gaseous particles, they are considered to move in a vacuum. This means we cannot model the viscosity of cytoplasm, which would intuitively lead to a lower velocity.

Another model proposed by [JHZG07], determines the displacement of a particle

described by Brownian motion as a function of temperature:

$$r2 = \frac{3kT}{3\pi\eta\alpha}t \tag{3.7}$$

Where r is the one-dimensional displacement, η is the viscosity of the given liquid the particle is submerged in, and $\alpha = 6\pi\eta a$, where a is the radius of the particle (which is assumed to be a sphere). This would then seem more suitable in terms of modeling the thermodynamic motion of the species of the devices. Although, much more precise models of motion of complex structures such as DNA strands and the finer *hydrodynamic interactions* between particles have been proposed in [AS13] and [GW09].

3.6 Systems Biology Markup Language (SBML)

A popular data representation of chemical reactions systems is SBML. Since it is based on the XML format, it is a standard of expressing a computational model with a wide range of applications. Generally an SBML file is read by a given parser and compiled into a data structure on which a simulation algorithm can be applied. The version, of which the parser for the project is constructed, is 3.1. The documentation for the whole language can be seen in [HBH+10].

Parsers for SBML do already exist ready for use, but only supporting specific platforms not including .NET. The frameworks that seem most supported is Python [pyt] and Java [jav] [lib]. For this reason, we are going to construct our own parser, in which we extract the relevant data for simulation. The parser is further discussed in Chapter 4.

Describing a chemical reaction system in SBML

The SBML language is quite extensive, both adopting syntax from XML and MathML and others, meaning that we will only look at the most essential parts of the syntax.
The main components are: a set of *compartments*, *species*, and *reactions*. A compartment specifies an environment, where a set of species react by a set of reactions, much like as described earlier. An example of the concrete syntax of a compartment is as follows:

```
...
<listOfCompartments>
    <compartment id="compartment1" spatialDimensions = "3.0" size="2000"/>
</listOfCompartments>
...
```

where the `id` is a unique identifier later used to reference it when declaring species and reactions, the `spatialDimensions` parameter denotes in how many dimensions the compartment is considered (in this case 3), and the `size` parameter denotes the physical size of the compartment.
A species can be declared by the following:

```
...
<listOfSpecies>
  <species id="Plac" compartment="compartment1" initialAmount="1"
      hasOnlySubstanceUnits="true"/>
  ...
  <species id="Plac_Lacl" compartment="compartment1" initialAmount="0"
      hasOnlySubstanceUnits="true"/>
</listOfSpecies>
...
```

where the `id` is a unique identifier later used to reference it in reactions, the `compartment` parameters is the identifier of the compartment it is in, the `initialAmount` parameter denotes the amount of the given species at the initial state of the chemical system, and the `hasOnlySubstanceUnits` denotes how the amount of the given should be interpreted in reactions, e.g. if true it should be an amount not depending on the size of the compartment. The reactions can then be specified as follows:

```
<listOfReactions>
  <reaction id="transcription" reversible="false">
    <listOfProducts>
      <speciesReference species="mRNA"/>
    </listOfProducts>
    <listOfModifiers>
      <modifierSpeciesReference species="Plac"/>
    </listOfModifiers>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <cn> 0.5 </cn>
```

```
          <ci> Plac </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
</listOfReactions>
```

where a reaction has a unique identifier `id`, the `reversible` denotes in which directions the reaction is allowed, i.e. if it is false, it describes a reaction of $A \rightarrow B$. It is then described by its list of products, and reactants or modifiers, and a kinetic law. The product and reactant/modifiers each have a reference to a species as described before. A modifier relates directly to the modifier arc proposed earlier for the SPN, i.e. a reaction in the form of $A \rightarrow A + B$. The kinetic law then describes the kinetic law of mass action in Equation 3.3. Here we have a `ci` parameter denoting the rate constant and the `ci` parameter denoting the species the rate function relies on.

This only describes a small part of the SBML syntax, in which the same chemical system can be described in different ways, depending on application it is meant for. And as described in Chapter 4, this means we will not cover the whole syntax including all possible XML and MathML constructs.
The standard for SBML is also in an ever changing state. This results in a limitation of the end product, since it will be deemed for continues maintenance, since the parser is custom made for the framework.

3.7 Implementation environment

Before transitioning to the design phase, an important aspect of the project, is also to investigate whether the choice of implementation technology has a role to play, once we start altering/modifying the implementation. Be it, by changing or extending the model or other.

A modular design and implementation of the framework is important, which in software engineering also is an important discipline to master. Achieving this highly relies on methodology, but also the choice of technology, depending on the experience which the implementer has within the given technology. A frequent and popular choice of technology i.e. programming language, in many cases of software implementation, resides in the object-oriented family.
Java is a good example of this - providing a wide range of mature and versatile tools. But as the implementation grows in size (lines of code), extending with features or refining the model often becomes cumbersome, often caused by high cohesion between the different components of the framework. Choosing another programming paradigm such as the functional, will provide much easier means of implementing a modular and extensible framework.

3.8 Summary

In this chapter we have identified the key components of the synthetic genetic devices, such that we are now able to understand and evaluate the behaviour resulted from simulations done later in Chapter 7.
In order to model such devices, they were introduced as general chemical reaction systems, introducing a broad selection of model and algorithms. The general method of ODEs was described in order to motivate a more sophisticated model, introducing stochastic elements. Gillespie's is considered the golden standard for simulating a stochastic chemical reaction systems, motivating further research into the statistical estimation of particle collisions. A link between this direct method and the ODE was identified as the model of increasing rate of reaction proportional to the population size of the relevant species. Which is the general mechanic of this basic model, i.e. with an increasing population the probability of two particle interaction increases. This argument will later be interesting to test in Chapter 7, by comparing the negative feedback device (in which a protein can only meet *one* promoter) with the and-gate device (in which two proteins of equal population size meet for reaction).

Chapter 4

Design

To give an overview of the tool developed, a component diagram is presented in Figure 4.1.

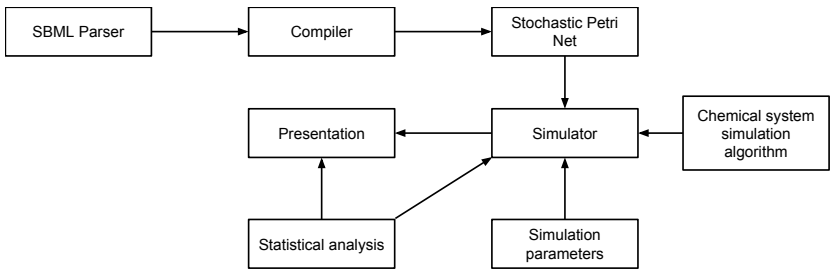


Figure 4.1: A component diagram of the entire system.

The tool should enable simulation of the genetic devices shown earlier in Figure 3.3 and 3.4. The process of doing so, should follow the flow presented in the activity diagram in Figure 2.1, thus satisfying the requirements stated in Chapter 2.

We see that an SBML parser parses an SBML file describing a genetic device, outputting a model which is then compiled into a stochastic petri net (SPN).

That is then used in a simulator, which inputs a given chemical systems simulation algorithm among other simulation specific parameters, such as duration and amount of simulations. The result of the simulation can then be analysed, e.g. by performing some statistical analysis. A presentation, be it a simple graph or animation, can then illustrate the simulation result to the user, which should provide some insight of mechanics of the given device.
These components can then be replaced or improved on, hence the discussed requirement of easy model refining or other improvements to the tool in general.

4.1 SBML Parser

The parser is constructed using the `FsLex` and `FsYacc` framework. As mentioned in Chapter 3, there are no suitable solutions for parsing SBML files currently available for the `.NET` framework. The chosen frame work will then provide us with the flexibility of constructing our own custom parser, though not covering the whole SBML specification.

An SBML file is first read into a *lexer*, recognising key words which are tokenized. For an explanation of the concrete syntax of SBML, please refer for Chapter 3. The abstract syntax tree (AST) then describes the structure of file containing terminal and non-terminal symbols. The grammar will be described in terms of presenting the most essential parts, leaving out details of smaller parts of it. The outputted AST consists of non-terminal symbols *Parts*, being either a list of compartments, species, or reactions. This part of the grammar is as follows.

$$\begin{aligned} \langle Part \rangle \quad ::= \; & \langle Compartments \rangle \\ | \; & \langle Speciess \rangle \\ | \; & \langle Reactions \rangle \end{aligned}$$

This means the order of which these parts are declared is of no significance. But in general, one or more compartments are first declared followed by their respective species and reactions.
A list of compartments is then denoted as the non-terminal symbol *Compartments* of the following grammar:

$\langle Compartments \rangle$ ::= STARTTAG COMPARTS ENDTAG $\langle CompMap \rangle$ STARTTAG END COMPARTS ENDTAG

Here, the list of compartments are enclosed by a specific sequence of tokens[1] i.e. `'<...> ...compartments...</ ...>`, which is the general XML notation for maintaining the hierarchical structure. The non-terminal symbol *CompMap* denotes the resulting map of compartments of the model outputted by the parser, later described in detail. The rest of the grammar should now be self-explanatory, in which compartments and species are defined by a number of parameters and identifiers, and each reaction has it kinetic law described, in which the parser can handle simple constructs such as '`Plac*0.1`', as seen in the concrete syntax in Chapter 3. A reaction also contains a list of reactants and products, in which the reactant can be a modifier relating directly to the modifier-arc described for the SPN.

⟨*CompMap*⟩ ::= ϵ
| ⟨*CompMap*⟩ ⟨*Compartment*⟩

⟨*Comparment*⟩ ::= STARTTAG COMPART ID EQ STRING ⟨*CompartParas*⟩ END ENDTAG

⟨*CompartParas*⟩ ::= ⟨*Name*⟩ ⟨*SpaDim*⟩ ⟨*Size*⟩ ⟨*Units*⟩ ⟨*Constant*⟩

⟨*Speciess*⟩ ::= STARTTAG SPECS ENDTAG ⟨*SpecMap*⟩ STARTTAG END SPECS ENDTAG

⟨*SpecMap*⟩ ::= ϵ
| ⟨*SpecMap*⟩ ⟨*Species*⟩

⟨*Species*⟩ ::= STARTTAG SPECIES ID EQ STRING ⟨*SpeciesParas*⟩ END ENDTAG

⟨*SpeciesParas*⟩ ::= ⟨*Name*⟩ COMPART EQ STRING ⟨*InitAmt*⟩ ⟨*InitConc*⟩ ⟨*SubsUnits*⟩ HASSUB STRING ⟨*BoundCond*⟩ ⟨*Constant*⟩ ⟨*ConvFact*⟩

⟨*Reactions*⟩ ::= STARTTAG RECS ENDTAG ⟨*RecMap*⟩ STARTTAG END RECS ENDTAG

⟨*RecMap*⟩ ::= ϵ
| ⟨*RecMap*⟩ ⟨*Reaction*⟩

⟨*Reaction*⟩ ::= TARTTAG REACTION ID EQ STRING ⟨*Name*⟩ REVERS EQ STRING ⟨*Fast*⟩ ⟨*CompartRef*⟩ ENDTAG ⟨*AgentList*⟩ STARTTAG KLAW ENDTAG ⟨*KinecticLaw*⟩ STARTTAG END KLAW ENDTAG STARTTAG END REACTION ENDTAG

[1]Please note, that when talking about tokens in the context of the parser, we are referring to the tokenized non-terminals, not tokens in an SPN.

The whole AST of the SBML file seen in Appendix A is rather large. But in order to illustrate its structure, a snippet of an AST produced is illustrated in Figure 4.2. Here, a *part* contains a map of compartments, in which one compartment resides with its respective parameters.

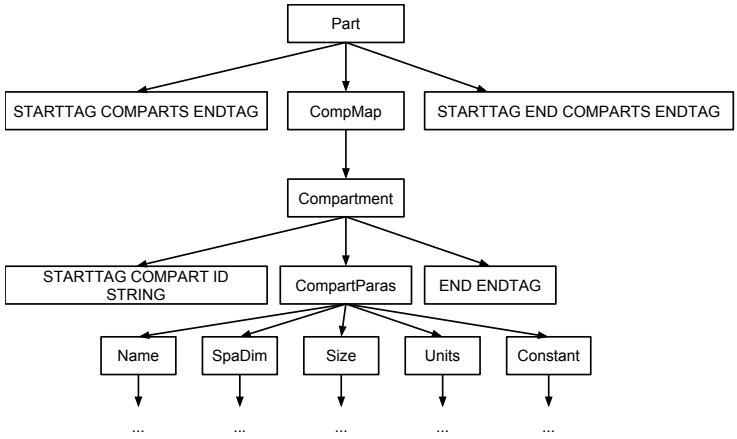


Figure 4.2: An example abstract syntax tree of a declaration of a compartment.

The parser outputs a model of the following type, seen in Appendix C:

```
type Part = | AllCompartments of Map<string,Compartment>
            | AllSpecies of Map<string,Species>
            | AllReactions of Map<string,Reaction>
            | Undefined

type Model = List<Part>
```

The underlying type declarations for `Compartment`, `Species`, and `Reaction` are omitted in this case.
The `Compartment` type describes the environment in general, such as defining custom units etc.
The `Species` type describes the different species in the system, such as declaring its name, what compartment it is in and its initial amount.
The `Reaction` type describes the set of reactions defining the device and the transitions of the later compiled SPN. An important note on this type, is its vulnerability towards changes in the SBML documentation, causing alteration in the parser, after which the `Model` type should be changed. For this, the type `Undefined` is included for managing artefacts not supported by the parser.

Adding this type, the process of extending the parser whilst keeping the compiler untouched should be easy, i.e. the compiler will ignore such even though the parser understands the concrete syntax of the given statement in SBML. Of course the compiler must eventually be extended, but in terms of developing the tool, this provides some independence between the modules.
The implemented parser is not complete in terms of understanding the full XML format of complex MathML constructs. This means that the parser itself should be subject to refinements and extensions, if needed. Pleae refer to the full grammar supported by the parser in Appendix D and lexer in Appendix E. And the full specification of SBML can be found in [HBH+10].

The main reason for not having the parser just output an SPN instead of the generic `Model`, is to leave the possibility of implementing a whole new data structure e.g. if we wanted to implement the probabilistic automaton described in Chapter 3.

An SBML file is parsed by utilizing the `ParserUtil` module, which has the functions listed in Table 4.1.

Name	Type	Legend
`parseString`	`string -> Model`	Reads a string (SBML file) and returns the model.
`parseFromFile`	`string -> Model`	Opens a file, and if it exists, uses `parseString` to read the file.

Table 4.1: Functions for parsing SBML in the *ParserUtil* module. Please refer to Appendix F.

4.2 Stochastic Petri Net

Before we describe the compiler, taking the `Model` as input and outputs and SPN, we need to describe how it is defined. Due to its many stages through the project, when experimenting with different data structures and models, a base type for an SPN is declared as following:

```
type Tokens<'a> = 'a
type RateFunction = float
type Transition = string * RateFunction
type Place = string
type Arc = | TransPlace of Transition * Place
           | PlaceTrans of Place * Transition
```

```
          | Modifier of Place * Transition

type SPN<'TokenCollection,'Token> = {
    marking : Map<Place,'TokenCollection>;
    transitions : Map<Transition,Arc List>;
    genTokens : int->'TokenCollection;
    optional : Option<Space>;
    tokensInPlace : Map<Place,'TokenCollection>->Place->int
    removeToken : 'TokenCollection->'TokenCollection
    fireRule : SPN<'TokenCollection,'Token>->Place list->Transition->Option<
        Space>
                    ->'Token option*Map<Place,'TokenCollection>
    addToken : Map<Place,'TokenCollection>->Place list->'Token->Map<Place,'
        TokenCollection>
    nextState : SPN<'TokenCollection,'Token>->float->SPN<'TokenCollection,'
        Token>
    }
```

The data structure keeps track of all the places by keeping them in a map, providing fast look-ups, with the place (i.e. name of the species) as the key. A marking is then contained within each place denoting how many tokens it has. A transition then has a name, i.e. the name of the respective reaction, and a rate function specified by its respective kinetic law. The transitions are then linked to the places by keeping them in map, having the transition as the key. An arc is then either pointing from a place to the transition or from the transition to a place. So when firing a transition, we simply have to work with just these two sub data structures.

This base type should then be extended by first declaring the necessary types of tokens and how they are collected - e.g. a list or array. The generic `genTokens` function is used when compiling the model outputted by the parser to generate the given type of token. The `tokensInPlace` function counts the tokens in a given place of the SPN, which differs in terms of how the tokens a collected.
The `removeToken` function removes a token from the specified type of token collection, e.g. if it was an array, a special function is then needed to maintain the integrity of the array.
The `fireRule` function is essential for simulation, i.e. it describes how transitions are fired. If for instance we want to describe the spatial model proposed by O.Maler, we want to try and find two neighbouring particles if the given transition has two ingoing places, instead of simply checking if the transition is enabled.
The `addToken` functions adds a token depending on the collection of tokens, much like the `removeToken` function. Lastly, the `nextState` function computes

the next spatial state of the chemical system presented by the SPN, given by the optional field `option`, which is to be ignored in the basic SPN with integer counter for tokens.

This enables the declaration of any kind of SPN, in terms of how we see particlesby the different model abstractions. But also in a technical sense when improving performance, e.g. when finding neighbouring particles. Auxiliary functions used for simulation are provided in the `SPNbase` module, which are listed in Table 4.2.

Name	Type	Legend
`genNextState`	`SPN<'a,'b>-> (float ->SPN<'a,'b>)`	Outputs a new SPN where the given function `nextState` is applied.
`state`	`SPN<'a,'b>-> Map<Place,int>`	Outputs the marking of the SPN, where tokens have been counted for each place.
`isOfInterest`	`Transition ->string list ->SPN<'a,'b>->bool`	Checks if the transition has outgoing places with the name of at least one in the list of strings.
`a`	`SPN<'a,'b>-> (string * float) list`	Outputs the propensity of each transition.

Table 4.2: Functions for the *SPNbase* module. Pleaser refer to Appendix G.

Space and Motion

When considering the positions of particles in a chemical system, we need to keep track of their coordinates. This is described by the following type:

```
type Coordinate = {x : float; y : float; z : float}
```

One could have extended this type to be flexible in terms of describing a coordinate in n-dimensions, but for the purpose of jumping straight to three dimensions, simulating in two dimensions is unnecessary.

The different SPNs used in this tool are then each constructed in the `SPNint`, `SPNlist`, `SPNarray` modules. They each have a function for initialising their respective SPN of the type `SPNbase`. The functions are listed in Table 4.3.

Name	Type	Legend
SPNint.makeSPNint	SPN<int,int>	Instantiates an SPN with integers as tokens with proper functions
SPNlist.makeSPNlist	Space -> SPN<Coordinate list, Coordinate>	Instantiates an SPN with coordinate lists as tokens with proper functions
SPNarray.makeSPNarray	Space -> SPN<Coordinate array, Coordinate>	Instantiates an SPN with coordinate arrays as tokens with proper functions

Table 4.3: Functions for the *SPNint*, *SPNlist*, and *SPNarray* modules. Please refer to Appendix H, I, and J.

The purpose of having *SPNlist* and *SPNarray* modules are to compare their performance when simulating the SPN, which is later evaluated in Chapter 5.

When extending the basic model for spatial dynamic, it was also determined that it would be suitable to simulate with realistic thermodynamic motion and other parameters. For purposes of experimentation, it would be suitable to test different kinds of motion: with constant speed, or with velocity vectors as defined in Equation 3.5 or 3.6. The type `Motion` is then defined as follows, such that we can alter between different kinds of motion.

```
type Motion =
    | Constant   of float
    | VelocityAtTemp  of (float -> float)
    | VelocityAtTempRad of (float -> float -> float)
```

How the noise vector, described in Chapter 3 about Brownian motion, is distributed would will also be subject to experimentation. For this, the type `Distribution` is declared, such that when displacements of particles coordinates are applied, the type of distribution is checked by matching the following type.

```
type Distribution =
    | UniformDist of ContinuousUniform
    | NormalDist of Normal
```

These types are declared in the `BrownianMotion` module, containing additional functions for generating noise vectors and applying the different types of motion, which are listed in Table 4.4.

Name	Type	Legend
`genNoiseVector`	`Ditribution -> Coordinate`	Generates a noise vector following the given distribution.
`applyMotion`	`Motion -> float ->float ->float`	Outputs the unit length of the velocity vector depending on the motion.

Table 4.4: Functions for the *BrownianMotion* module. Please refer to Appendix K.

The spatial model considered in this project, inspired by the individual dynamic model proposed by O.Maler, is defined by the type `Space`. It contains all necessary fields for simulating the different kinds of `Motion` and `Distribution`, but also how large the environment (cell) is, the radius of the particles, and how much kinetic energy is lost when a particle bounces into the barrier of the environment.

```
type Space = {
    size : float;
    radius : float;
    temperature : float;
    motion : Motion;
    distribution : Distribution;
    elasticity : float}
```

The `Space` type is declared in the *Space* module, e.g. containing functions for finding neighbouring particles and moving particles. These functions are listed in Table 4.5.

Name	Type	Legend
`boundary`	`float -> Space -> float`	Outputs a legal displacement, that is within the boundaries of the space.
`genMove`	`Space -> float -> Coordinate`	Outputs a new displacement vector for a particle, which depends on the given motion type that is used.
`inRange`	`Coordinate -> Coordinate -> Space ->bool`	Checks whether two coordinates are in range of each other, given by a radius in the space type.
`findneighbours`	`Coordinate list ->Coordinate list ->Space -> Coordinate list`	Outputs a random pair of coordinates, one partner from each list, that are neighbours.
`findneigbhours-array`	`Coordinate array ->Coordinate array ->Space -> Coordinate array`	Outputs a random pair of coordinates, one partner from each array, that are neighbours.

Table 4.5: Essential functions in the *Space* module. Please refer to Appendix L

A clear distinction between this module and the `BrownianMotion` model must be made, in terms of the responsibilities. In Table 4.5 we see the `genMove` function, which does not generate the displacements of the particles, applies the moves generated by the motion module. Having a clear means of delegation here, enables ease of possible extension of the thermodynamic model.

When moving a particle, a displacement vector is generated depending on the before mentioned types of motion and noise vector distribution. If a generated move would cause an 'illegal' position, i.e. out of the boundaries of the cell, the `boundary` function mimics a bouncing of the particle on the cell membrane, where the particle loses a given amount of kinetic energy given by the `elasticity` factor. This effect is illustrated in Figure 4.3. If the speed of the particle is so great, that upon bouncing back to another[2] illegal position is produced, the function recursively bounces the particle back again until its displacement factor has been exhausted.

[2]Which turns out to be the case, when using the thermodynamic model described by Equation 3.5

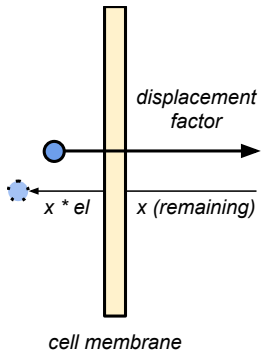


Figure 4.3: An illustration show a particle moving past the boundary of the cell membrane.

It is also important to note that the functions for finding neighbouring particle both output a pair which is chosen at random, eliminating any priority of which particles are added and removed from places in the SPN. Doing so, we achieve the same neighbour evaluation as in Algorithm 2 for individual spatial dynamics proposed in [MHML14].

We can now, by utilizing these types, initialise an empty SPN, for which a `Model` will be compiled onto it. Let us consider the following example of initialising a space environment.

```
let kB = 1.3806488e-23 //Boltzmann's constant
let particleMass = 6.4e-22 //realistic mass of protein in grams
let unitScale = 10.0e+9 //for nanometer

let space = {
    size = 2000.0;
    radius = 4.0;
    temperature = 298.0;
    motion = VelocityAtTemp(fun temp ->
        (sqrt((3.0*kB*temp)/particleMass))*unitScale)
    distribution = NormalDist(normalDist);
    elasticity = 0.5}
```

Here we have defined a space describing a cell of 2000.0 unit length in nanometres, the radius of each particle in the device is set to 4.0, and the temperature is 298 Kelvin. The motion is set to that of Equation 3.5, and particles should lose half their kinetic energy when they bounce into the cell membrane.

A instance of `SPNbase` with arrays and coordinates can then be declared as the following:

```
let spnarr = makeSPNarray space;;
```

Doing so, leaves the marking and transitions empty. This is because the information of the chemical reaction system is yet to be compiled onto it, which is the next step.

4.3 Compiler

The information of a model, outputted by the parser, is then inserting in an empty SPN of type `SPNbase`. The `SPNbaseCompiler` module provides the necessary function for this, as seen in Table 4.6.

Name	Type	Legend
`compileModel`	`SPN<'a,'b>-> Model -> SPN<'a,'b>`	Compiles the given model into the empty SPN, initialising its marking and transitions.

Table 4.6: Function for the *SPNbaseCompiler* module. Please refer to Appendix M

An important note on the compiler is that in SBML specification, species and reactions are linked to a compartment through identifiers. Which the compiler does not take care of, since the models described in SBML used for this project do not describe more than one compartment. The purpose of having different compartments, is if one wants to declare several reactions system within the same SBML file, which in terms of the genetic devices, is not of importance. But if this should be added, one would simply only have to extend the compiler module, such that it either creates an SPN instance of each compartment or disjoint SPNs within the same instance of an SPN. We would then either simulate the different SPNs separately or together, depending on the nature of the chemical reactions system of course.
Continuing the example, we can compile the SPN as following:

```
let m = ParserUtil.parseFromFile "...\casestudy.xml";;
```

```
val m :  Model =
    [AllCompartments (map [("compartment1", name = null;
    spatialDim = Some 3.0;
    size = Some 2000;
    units = null;
    constant = null;)]);
    AllSpecies(map[("LacI", ...);("Plac", ...);("Plac_LacI",
    ...);("mRNA",...)]);
    AllReactions
    (map[("decay_LacI",...);("decay_mRNA",("regulation",...);
    ("repressed_transcription",...);("transcription",...);
    ("translation",...);("unbinds",...)])]

let spn = compileModel spnarr m;;
val spn1 :  SPNbase.SPN<Coordinate.Coordinate array,Coordinate.Coordinate>
    ={marking =map[("LacI", [||]); ("Plac", [|x = 1000.0;y = 1000.0;z =
    1000.0;|]);
    ("Plac_LacI", [||]);
    ("mRNA", [||])];
    transitions =map[(("decay_LacI", 0.0012), [PlaceTrans
    ("LacI",("decay_LacI", 0.0012))]);
    (("decay_mRNA", 0.0058), [PlaceTrans ("mRNA",("decay_mRNA", 0.0058))]);
    ...];
    ...}
```

This gives us the full SPN, where tokens are considered by their coordinates stored in an array. The environment illustrated here, is that of the negative feedback device mentioned in Chapter 3 and listed in Appensix A. Here we see e.g. the promoter is the only initial particle in the cell, residing in the centre of the cell.

4.4 Simulator

When the given chemical reaction system is compiled into a given SPN, it is ready to be simulated. The module `Simulator` inputs a simulation algorithm, a data structure (in this case an SPN), a list of particles names that are of interest of the simulation, an interval of which snapshots should be taken during the simulation(s), a duration for each simulation, and how many simulations that should be run. This function is listed in Table 4.7. The types used for the simulations are as follows:

```
type Data = (float*float) list list
type MinMaxData = ((float*float) list*(float*float) list) list
type Snapshots<'datastructure> = 'datastructure list
type SimulationResult<'datastructure> = ((Data * MinMaxData) * Snapshots<'
    datastructure>) * float list list
```

The `Data` type denotes a list of data points describing the concentration of a type of species in the cell at a given time step, of which there is a list for each type of species that is of interest.

The `MinMaxData` type denotes a list of data points for each species of interest, which is the 'maximum' and 'minimum' of all the simulations done. And by that, a 'maximum' of a simulation is global maximum measurement (yielding the highest integral) of each time step and vice versa for 'minimum', later used for presentation. This could of course have been done later in the statistical analysis itself. But by declaring this type and evaluating the maximum and minimum during simulation, we can easily distinguish between these two types of data later e.g. during presentation.

The `Snapshots` type denotes a list of states of a given data structure, be it an SPN or other, at each given time interval, later converted into text files and plotted and animated in Matlab.

Lastly, the `SimulationResult` type denotes the result of a simulation, containing the data of accumulated average behaviour of the given device, the maximum and minimum measurement, the snapshots, and a list of the concentrations measured for at the last time step of each particle of interest, later used for calculating an interval of mean concentration in a given level of confidence.

Name	Type	Legend
`simulate`	`'algo ->datastructure -> string list ->'snapinterval ->'duration -> 'simulations ->int -> SimulationResult<'spntype>`	Inputs a given chemical system given by a data structure by a given simulation algorithm and outputs a simulation result.

Table 4.7: Function for the *Simulator* module. Please refer to Appendix N

The simulations are parallelised. For this, a functionality for managing how many simulations are done per task is provided. Providing the flexibility of achieving optimal performance, depending on the amount of threads provided

by the given hardware contra the overhead of constructing the task list itself.

4.5 Chemical system simulation algorithm

The different kinds of simulation algorithms used for this project are implemented as individual modules as listed in Table 4.8

Name	Type	Legend
`gillespie`	`SPN<'a,'b>-> string list -> float ->int -> System.Random -> (SPN<'a,'b>* float) list * (float * Map<string,int>)[]`	Inputs an SPN, parts of interest,a snapshot interval, duration,a random number generator, andouputs the snapshots and markings at each time step. Calculated using Gillespie's direct method.
`spatial`	`SPN<'a,'b>-> string list -> float ->int -> System.Random -> (SPN<'a,'b>* float) list * (float * Map<string,int>)[]`	Inputs the same as `gillespie`, but takes the particles positions into account.

Table 4.8: Functions for the simulation algorithm modules: *Gillespie* and *SpatialGillespie*. Please refer to Appendix O.

The main difference between the two is that one is the general Gillesipe algorithm, and the other is an extension of such inspired by the individual spatial model proposed by O.Maler. These are later compared in the experiments in Chapter 7.

Continuing the example, we can now initiate a simulation as follows:

```
let duration = 4000;;
let simulations = 100;;
let snapInterval = 1.0;;
```

```
let partsOfInterest = ["mRNA";"LacI"];;

let spatialsims = simulate spatial spn partsOfInterest snapInterval
    duration simulations;;

val spatialsims :SimulationResult<SPN<Coordinate.Coordinate
    array,Coordinate.Coordinate> *float> =((([[(435.6666642, 275.6831683);
    ...(425.1273841, 276.2475248); ...];[(435.6666642, 3.217821782);
    ...(425.1273841, 3.297029703); ...]],[([(395.8935861,
    364.0);...(385.0487857, 369.0); ...],[(492.4948919,
    154.0);...(482.1576449, 152.0); ...]);([(395.8935861,
    4.0);...(385.0487857, 4.0); ...], ...); ...]), ...), ...)
```

Here we get the result of 100 simulations, where each simulation has run the `spatial` function for simulating the spatial model. Each simulation then stops when 4000 seconds have elapsed, given by the time step calculation in Gillespie seen in Algorithm 1. About 1000 snapshots have then been taken, which are later seen as 'frames' of an animation. The snippet of the result above leaves out a lot, but we can see that the average concentration of the LacI protein at the last time step was 275.68 and 3.22 for mRNA. We also see that there is a high variance of the LacI concentration given by 364 and 154.

4.6 Statistical analysis

In order to evaluate the results the `Statistics` module provides functions for calculating a confidence interval of a mean. But auxiliary functions such as finding the coordinates that define the out limits of the device, later used for animations and graph representation.

The confidence interval calculated describes an interval of means, which with a significance of 5% can be within. This module would be the main focus, if the tool would to be extended in terms of the requirements of analysis done on synthetic biology described in Chapter 1.

The interval can then be calculated as follows:

```
let interval = confidenceInterval (snd spatialsims) 0;;

val interval :  float * float = (265.445203, 287.714797)
```

Name	Type	Legend
`confidenceInterval`	`float list list ->int -> float * float`	Outputs the confidence interval of a list of measurements at the same time step of several simulations. At is calculated at a significance of 5%.
`findMinMaxCoords`	`Snapshots< SPN<'Coordinate [],'a»-> float * float * float * float * float * float`	Outputs the minimum and maximum x,y, and z value of the provided snapshots. Later used for defining scatter plot dimensions in animations.
`globalMax`	`Data -> float * float`	Outputs the maximum x and y value of the provided data. Later used for defining plot dimensions.

Table 4.9: Functions for the *Statistics* module. Please refer to Appendix P.

So when we later, in Chapter 7, compare the behaviour of another device under different conditions, we can with a significance of said 5% say they are the same, if the resulted average concentration is within the interval calculated for a control.

4.7 Presentation

The data is now ready for presentation. For this, the tool provides two options: generating a simple `Gnu`-plot or extraction of the data, such that it can be imported into Matlab for 3D scatter plots and animations. Although the simple plot will provide knowledge on the general behaviour of the given device, some valuable insight on the movement of the particlea can be gained by generating animations of them. The `DataWriter` module provides the necessary functions for saving snapshots of a given simulation, listed in Table 4.10.

Continuing the example, where we have the result of a given simulation, we can save the snapshots as follows:

```
let snapshots = snd (fst spatialsims)
```

Name	Type	Legend
`saveValue`	`'a ->string ->unit`	Stores a value of given type 'a at the given path.
`restoreValue`	`string ->'a`	Restores a value of type 'a. Must be annotated.
`linesTofile`	`string -> seq<string> ->unit`	Write the sequence of string into the file in the path.
`saveSnapshots`	`Snapshots-> int ->int -> ->Space -> float->float -> float * float * float * float * float * float -> unit`	Stores snapshots of a simulation as text files at a directory. Note: change directory.

Table 4.10: Functions for the *DataWriter* module, used for storing data and extracting data for animations. Please refer to Appendix Q.

```
let boundaries = findMinMaxCoords snapshots
saveSnapshots snapshots duration simulations space maxX maxY boundaries;;

val it :  unit = ()
```

The `boundaries` will be used to set the axis of the scatter plot.
For creating the simple plot the `Plotter` modules provides the function for plotting the result, where one has to declare legend titles and colors, and specify if the maximum and minimum data should be shown. The function is listed in Table 4.11.

Name	Type	Legend
`plotdata`	`SimulationResult<'a>-> string list ->Color list -> bool ->unit`	Plots the simulation result with minimum and maximum as optional. A list of titles and colors are given for the legend.

Table 4.11: Function for the *Plotter* module. Please refer to Appendix R.

Plotting the data can then be done as follows:

```
let titles = ["avg-LacI";"avg-mRNA";"min-LacI";
              "max-LacI";"min-mRNA";"max-mRNA"]
let colors = [Color.Red;Color.Green;Color.DarkGray;
             Color.DarkBlue;Color.LightGreen;
             Color.LightGreen]

plotdata allSpacialsimulationsThermo2 titles colors true;;
val it :  unit = ()
```

Showing the graph in Figure 4.4.

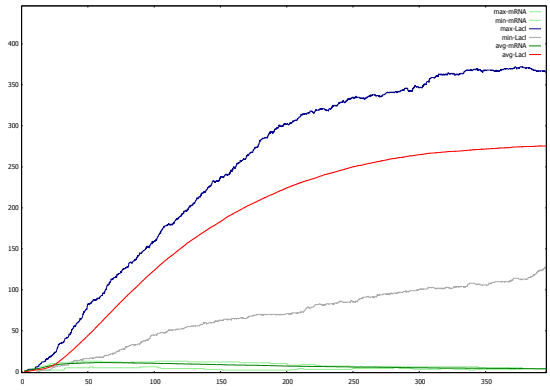


Figure 4.4: A plot of the example simulation done in this chapter.

Inspired by the neat 3D visualisations done in [dHCKMK13]. Animations are done in Matlab, please refer to the Matlab-script in Appendix S. In short, it reads two files for each snapshot taken during a simulation containing the information used to plot the concentration of LacI and another for a scatter plot showing the position of all the particles. Please refer to the experiment results in Chapter 7 for examples of these.

4.8 Summary

The described modules should now enable the fundamental work flow both described in the activity diagram in Figure 2.1 and framework in Figure 1.2. A simulation is started by the user through a script of maybe a graphical user

interface, in which a data structure reflecting an SBML file is compiled. The simulation then starts and runs a given number of times, at each time step of said simulation the state of the environment is changed, which at the end is return as the result.

Looking back at the component diagram in Figure 4.1, the compiler module could for instance have just been a sub module of the SPN module, resulting in fewer components, but this would complicate possible modification and replacement of a data structure, since the compiler for the data structure would also have to be reworked.
One could also propose that the `Statistics` module should somehow be incorporated into the `Simulator` module, referring back to the note on performance gain on procedural analysis during simulation. But due to wide range of different analysis that are possible, it was chosen to keep this module separate, thus achieving the behaviour described in Figure 2.1.

Chapter 5

Implementation

In this chapter we will give a brief overview of the implementation of each module, in terms of non-trivial choices that have been taken especially in the context of performance. One of the main challenges when implementing the toll was to try and satisfy the requirement stated in Chapter 2 for the simulator - that exploration of different techniques used for performance optimisation should be done.
For this, some key features within the F# framework have been utilised for both measuring and improving performance. The main goal here, wa to run simulations in parallel instead of sequentially.

Later when the spatial model was implemented, a performance bottleneck was introduced in terms of finding neighbouring particles, in which the same procedure was applied.
Neighbour searching in a dynamic system, i.e. ever moving particles, is a subject of its own, and will be shortly discussed in terms of what solutions are possible.

5.1 Parser and compiler

Implementing the parser and compiler was done, as mentioned in Chapter 4, by utilising the `FsLex` and `FsYacc` frame work for constructing a scanner and parser. After which the parsed model from the given SBML file is compiled onto a stochastic petri net. The structure of the implementation itself for both the parser and compiler, seen in Appendix E, D, and M, follows the basics concepts for constructing any given parser and compiler.

5.2 Simulator

Before discussing the implementation of the different kinds of stochastic petri nets, the simulator was first optimized.
Running an increasing amount of simulations is the most straightforward bottleneck present within the tool, thus exploiting the simple implementation overhead needed for doing a task parallel computation was highly motivated. We can see each simulation as a task that has to be done, to which a thread can be delegated.

Let us consider two different ways of computing the simulations:

```
let tasks = 20
let duration = 4000
let iterations = 160
let simulateFor time =
    for i in 0..iterations do gillesipe spn duration []

let rec naiveSimulate i max =
    if i < max then simulateFor resultSize::naiveSimulate (i+1) max
    else [simulateFor duration]
let sequential = naiveSimulate 0 tasks

let simulationtasks = Array.init tasks (fun _ -> duration)
let parallel = Array.Parallel.map simulateFor simulations
```

Here we are computing 3.200 $(20 * 160)$ simulations each running until 4000 seconds have passed in simulated time. The first `sequential` function computes the simulations recursively until its finished and stores the results in a list. The second `parallel` utilised the `Array.Parallel` library in which an array of tasks is first initialised, on which the `Array.Parallel.map` functions maps over the

task array. The results for both of the simulations are stored in a list. The performance of these are shown in Figure 5.1.

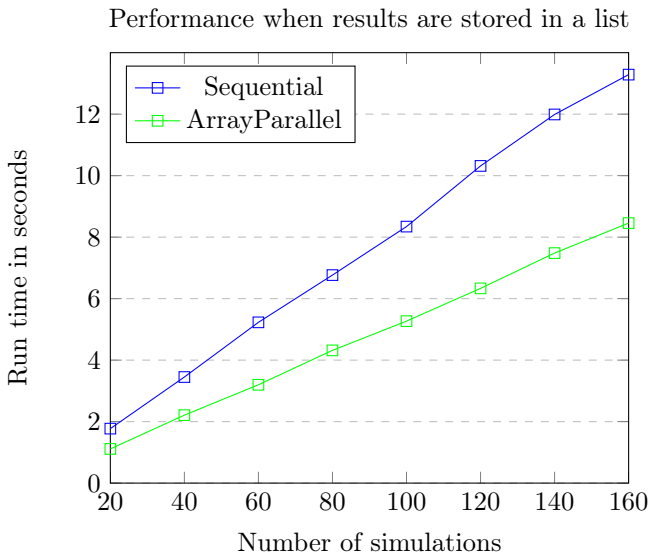


Figure 5.1: A plot showing the running times of the sequential and parallel simulations, where the result of each simulation is stored in a list.

Although we do see some performance gain, the simulations are run one a system with four cores, in which we could have expected to have gained performance by a factor of four. Using lists for storing the results of the simulations will, with the rather large amount of data stored in them, cause poor cache performance. This is mainly due to the `List` type describing a linked list, which provide good insertion and removal time complexity but poor memory complexity - $O(2n)$ where n is the amount of data points. To improve on this, storing the simulations results in an array which, having a memory complexity of $O(n)$, was implemented.

Let us consider the following:

```
let result =
    Array.init resultSize (fun _ -> (0.0,[("",0)]))
let simulateFor time =
    for i in 0..iterations do gillespie spn3 resultSize result
```

When we then run the same functions again, we get the performance shown in Figure 5.2.

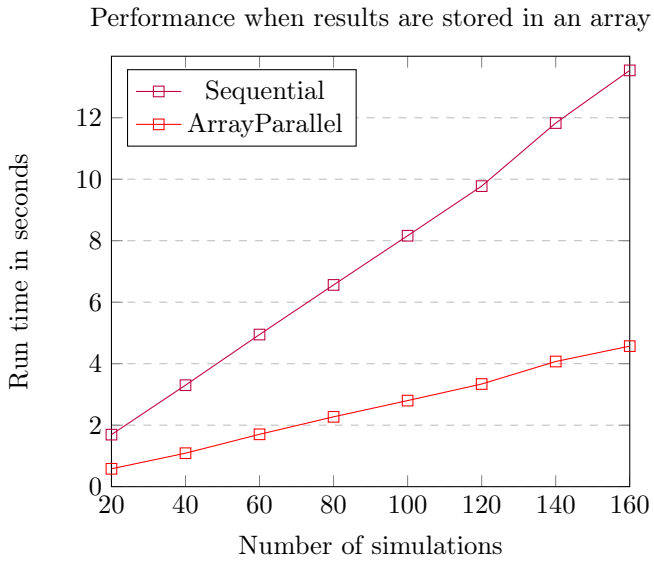


Figure 5.2: A plot showing the running times of the sequential and parallel simulations, where the result of each simulation is stored in an array.

Here we clearly see, that we are getting closer to the desired speed-up of factor four. The end result of the simulator can be seen in Appendix N, in which the simulations are computed in parallel following the technique described before. The user can specify maximum allowed number of simulations per task, which purpose is discussed in the section about stochastic petri nets.

5.2.1 Generating random numbers in parallel

A major, an later discovered, limitation of the popular tool for generating random numbers `System.Random` is its missing thread safety property. If we ran parallel simulations using this, the object created would eventually break by continuously returning a zero-value. A solution to this would be to wrap this object around a thread safe environment e.g. by implementing a semaphore. But the popular `MathNet` tool already provides a wrapped random number generator which is thread safe. As seen in the `Motion` type described in Chapter 4, uniformly and normally distributed numbers are generated using the `MathNet.Numerics.Distribution` libraries.

5.3 Presentation and statistics

Different kinds of tools were used for presentation, a `GnuPlot` type provider is used for generating in-tool graphs showing the results of the given simulation(s). The `Plotter` module, Appendix R, was implemented enabling different kinds of plots, i.e. optional minimum/maximums presentation and custom colours and titles for legends.
For computing the animations, MatLab has been used, where the data of a simulation is extracted by the `DataWriter` module Appendix Q. Data can be stored and restored, in which it is important to note that one should provide type annotation once restoring data from a file (which could otherwise corrupt the data file).
The `MathNet.Numerics.Statistics` library is used in the `Statistics` module for calculating the interval of confidence for a mean. It should be noted, that the function does not support a variable level of confidence, which is set to be 5%, hence the critical value of 1.96 seen in Appendix P.

5.4 Stochastic petri net

When introducing space into our model, we also increase the computational requirements for the simulations. When finding a pair of particles that are close enough for reaction, both the data structure in which they are stored and the search algorithm affect the performance. There exists a wide range of solutions for fast neighbouring searching dynamic systems, where the data structure has to be updated in terms of particle positions at each time step.

Let us consider two different ways of instantiating a simulation:

```
let spnlist =
    simulate spatial spnlist partsOfInterest snapInterval duration
        simulations

let spnarray =
    simulate spatial spnarray partsOfInterest snapInterval duration
        simulations
```

The difference between the two SPNs, as described in Chapter 4, is how we store the coordinates of the particles in an array, we can perform the neighbour search in parallel. This is done as following, in the `Space` module:

```
let neighbours =
    tokens2 |> Array.Parallel.choose
        (fun t -> if inRange ta t space then Some(t) else None)
```

Before doing so, we first ensure that `tokens2` is the longest of the two arrays in which we must find a pair. This avoids the common case, where we only have very few particles of one type that can react with a larger amount of another type.
The performance gains are listed in Table 5.1.

Name	Motion	Run time
`spnlist`	`Constant(0.0)`	19.691 s
`spnlist`	`Constant(6.0)`	41.981 s
`spnlist`	`Thermo1`	1797.226 s
`spnarray`	`Constant(0.0)`	17.695 s
`spnarray`	`Constant(6.0)`	18,395 s
`spnarray`	`Thermo1`	237.488 s

Table 5.1: A table of run times, where neighbours are first searched for sequentially `spnlist`, and in parallel `spnarray`.

20 simulations were done for each, where the simulator was allowed a minimum of 10 simulations per task (i.e. a task list of two tasks consisting of 10 simulations each), potentially leaving cores free for the parallel call for neighbour search. Allowing the simulator to use an unlimited amount of resources/threads, the gains of this improvement will either be small or actually decrease performance, since threads will be sleeping more. This illustrates the need for the flexibility needed in the simulator, in terms of task management.
The first simulations, where the particles do not move (hence `Constant(0.0)`), were done as a control and see the high variance in run time of the spatial model itself. The speed of the particles were then increased, ending with the thermodynamic motion described by Equation 3.6 in Chapter 3.
When increasing the number of simulations per tasks, when using the parallel neighbour search, close to no speed-up was achieved as seen in Table 5.2. This indicates that once a parallel simulation task exhausting the thread pool, later calling parallel computation provides no gains in terms of run times.

Tasklist	Run time
`[|10|]`	314.485 s
`[|5;5|]`	324.505 s
`[|3;3;4]`	276.649 s
`[|2;2;2;4]|`	292.829 s
`[|1;1;1;1;1;1;1;1;1;1|]`	287.858 s

Table 5.2: A table of run times, where the maximum simulations per task is decreased.

A major improvement that could be done to the current implementation of the `SPNarray`, in Appendix J, is to improve the amortised cost of inserting and removing tokens. The current solution simply inserts a token by using the `Array.append` function, and removes a token by create a whole new array where the given token is not present. This could be improved by describing an array of type `Coordinate option array` in which arrays are doubled in size, when the array is full of `Some(...)` instances of tokens.

5.4.1 Faster neighbour search

A solution for finding neighbouring particles in a static system, would be to construct a kd-tree of coordinates. In short, a kd-tree is a k-dimensional tree containing the sorted coordinates, e.g. each a level for the x, y, and z-values for 3 dimensions. By doing so, it effectively partitions the space in sub-areas. This provides fast range queries, average case $O(log\ n)$ where n is the amount of given type of particle, i.e. finding a set of coordinates that are within a given radius. But utilising a kd-tree for storing coordinates in a dynamic system, changing the positions of the particle at each time step, we would have to re-construct the entire tree, resulting in an amortized cost of $O(m\ log\ n)$, where m is the total number of particles, for reconstructing the tree. This is rather ineffective compared to $O(m)$ time for applying moves provided by either the list or array solution.
Solutions providing optimized data structures that adapt the concept of the kd-tree are found in the following:

- [DRF12] proposes a packed memory array (PMA), where particles are sorted by which 'cell' they belong to, i.e. much like the sub-areas of the kd-tree. Some indexes are then left empty to achieve and $O(log\ n)$ amortised cost of moves i.e. updating their positions in each time step.

- [ABRS12] proposes a list for adaptive resolution particles simulations, improving performance for simulating system in which particles have different reaction radii. The varying sizes of the species in the devices; mRNA strands and complex proteins, are not included in the model proposed in this thesis. But adapting this solution into the current model would be beneficial, and interesting to see if it had an effect on the dynamic behaviour of the devices. Though it should be noted, that this data structure does not provide as fast updates as in the PMA.
- [GW09] proposes an efficient data structure for computing hydrodynamic (HI) interactions between spherical particles, as described in Chapter 3. In physics, when computing HI a diffusion tensor is constructed with a runtime of $O(n^3)$. So if we later wanted to include HI into our model, we would introduce another bottleneck. An algorithm is then proposed, achieving an optimized runtime of $O(n^2)$.

5.5 Summary

The premise for showing the techniques used for improving on performance, was to show how little work is needed in order to speed up code in F#. We saw that by changing just a few lines of code, we could easily exploit the multiple cores available to us, which is a common set-up as of writing this report. We saw a performance gain by a factor of four - proportional to the amount of cores available when computing simulations in parallel. And an even greater factor when finding neighbouring particles in parallel.
Though it should be noted, that a task potentially utilising all threads leaves no more room for later parallel computations during run time. This means, that the speed-up of parallel simulations is no longer present once we parallelised the neighbour search. So depending on the device that is simulated, the gains of parallel neighbour search were not as prominent as expected. Although, storing tokens in arrays does not only provide parallel computation capabilities, but is also much more memory efficient, as we saw in the simulator section in this chapter. Let us consider a device that produces a high amount of a given species during a simulation. The list-solution would then potentially run out of memory much sooner than the array-solution, hence the doubled memory complexity of a list compared to an array.

Further improvements in terms of performance were described in terms of alternative data structures and algorithms, which will also later be summarised in the conclusion, Chapter8.

Chapter 6

Tests

Testing the entire implementation has not been of focus during the project. But a few test of essential components of the tool will be proposed in this chapter. The main purpose of doing this, would be to confirm the requirements listed in Chapter 2, thus concluding the software engineering aspect of this thesis.

6.1 Test overview

Testing a software system can be done in numerous ways, but they must be conducted such that they test both the functional- and non-functional requirements. One way is to do white- and black-box tests. White-box tests inspect the inner workings of the software, whilst the black-box tests cover the functionality of the software with no knowledge of how the given output is generated.

When testing the tool against the requirements, the method known as static black-box testing is used [Pat06]. The general process of this is to form a test case, be it a use-case of user scenario. The test should then be conducted with both input that is known to be correct and incorrect, after which the output is carefully evaluated. Different techniques for such tests are e.g. *pairwise testing* or *decision tables*.

6.1.1 Parser

The purpose of testing the parser is to see, if can parser the SBML files describing the devices in Appendix A and B. The parser does not cover the whole SBML specification, mainly due to its extensive documentation and flexibility in terms of defining a chemical reaction system.
The main process was first to see if tokens were correctly generated, by producing printouts when they were pattern matched by the lexer. The model outputted by the parser was then inspected, by comparing the generated F# types with the model described in the file.

6.1.2 Compiler

The compiler was first tested by constructing a basic instance of the `Model` type, containing no species etc. It was then expected that the compiled data structure would contain empty collections, i.e. remain untouched by the compiler. The following illustrates such test:

```
> let m:Model = [];;
val m :  Model = []

> let spn = makeSPNint;;
val spn :  SPN<int,int> = {marking = map [];transitions = map [];...}

> let compiledSPN = compileModel spn m;;
val compiledSPN : SPN<int,int> = {marking = map [];transitions = map
    [];...}
```

To see if a simple model describing a reaction system of $A \rightarrow B$ would e.g. produce a stochastic petri with still an empty marking, but with the correct collection of transitions, was also important to test.
Lastly, an outputted model from the parser was then used to compile its reflected SPN. Again, the evaluation of the result was done by carefully inspecting it for correct marking and collection of transitions. It was also important the we maintained the correct propensity functions in the transition, such that the simulated dynamic behaviour is correct in terms of the given parameters.

6.1.3 Data structures

The key feature of the SPNs is that they keep track of the global state of the given chemical reaction system it describes. For this, it was important to test if the `fireRule` function and its auxiliary functions for adding and removing tokens were as intended. An SPN having two places p_1 and p_2 connected by a transition t with the initial marking $m = [(p_1, 1), (p_2, 0)]$, then served as a base case for this type of test. By firing the transition it was then expected to remove the token in p_1 and add it to p_2 such that $m = [(p_1, 0), (p_2, 1)]$. Continuing the test in the section before, we can do as follows:

```
> let spnwithmarking =
{spn with marking = Map.ofList [("p1",1);("p2",0)]};;

val spnwithmarking :  SPN<int,int> ={marking = map [("p1", 1); ("p2",
    0)];transitions = map [];...}
> let spnwithtrans = {spnwithmarking with transitions = Map.ofList [(("t"
    ,0.0),[PlaceTrans ("p1",("t",0.0));TransPlace (("t",0.0),"p2")])]};;

val spnwithtrans :  SPN<int,int> ={marking = map [("p1", 1); ("p2",
    0)];transitions =map[(("t", 0.0),[PlaceTrans ("p1",("t", 0.0));
    TransPlace (("t", 0.0),"p2")])];...}
> let firedspn = fire spnwithtrans ("t",0.0);;

val firedspn :  SPN<int,int> ={marking = map [("p1", 0); ("p2", 1)];...}
```

Later more complex models were tested during the experiments themselves, which can be seen and evaluated by the results presented in Chapter 7. It should be noted, that in practice the *SPNlist* module was not used because of the described model extensions and performance gains provided by *SPNarray* in Chapter 5.
The spatial model defined by the modules *BrownianMotion* and *Space* both contain critical functions such as the `boundary` function ensuring that particles stay inside the cell. The premise of this function is described in Chapter 4, and can be tested in three cases; 'moves' being legal, illegal, and illegal whilst causing recursion as follows:

```
> let space = {size = 100.0; ... }
let case1 = boundary 110.0 space
let case2 = boundary 90.0 space
let case3 = boundary 2000.0 space;;
...val case1 :  float = 40.0  val case2 :  float = 90.0
    val case3 :  float = 31.25
```

The correctness of spatial model implemented is supported by the animations done in Chapter 7, showing particles moving within a restricted area with defined motion.

6.1.4 Simulator and Simulation algorithms

One could for instance test the simulation algorithms by means of conducting a pairwise test - by testing all possible input pairs, or test the simulator through a decision table by means of different parameters described in a decision table. An example of such can be seen in Table 6.1.

Conditions	**Rule 1**	**Rule 2**	**...**	**Rule n**
snapinterval $<$ duration	T	F	...	F
partsOfInterest exists in data structure	T	T	...	F
duration > 0	T	T	...	F
maxTasks > 0	T	T	...	F
Action			...	
Simulation is possible	T	T	F	F

Table 6.1: Example of a decision table used for testing the simulator with different parameters.

White box test were done when implementing the different simulation algorithms, by conducting a branch test, [Kha11], much like the decision table. Different parameters were then used to reach possible outcomes of the algorithms. One being the duration of the simulation, and another being the data structure reflecting the given device. A major vulnerability exposed by doing so, was the lacking robustness of the Gillespie's direct method and also the spatial model algorithm, in terms of possible input creating infinite amounts of a given particle resulting in a memory exception being thrown.

6.1.5 Presentation and statistics

Since the correctness of the presentation done relies much on the libraries utilised - `GnuPlot` and Matlab, extensive testing would not be necessary compared to the rest of the system. Although, it was important that the correct axis and legends

were constructed, which would be crucial to the outcome of the experiments if were done incorrectly. This was done by following the same procedure as for the rest of the system, i.e. starting with a base case followed by real examples.

6.2 Summary

This chapter should serve as an outline for conducting tests for the tool, if later needed. The general process of the tests follow the pair-wise testing method, with carefully chosen parameters covering both a base and special cases.
During implementation tests were done but not documented, which should have been done in hindsight, thus improving the overall quality of the documentation.

Chapter 7

Experimens and results

The purpose of this chapter is to catalogue the experiments that are the basis for the evaluation and analysis of the different models. The results of these experiments will then each be listed with a hypotheses given by detailed description and the expected result, parameters, results, and evaluation. After each experiment a rational will be given, motivating the next iteration of experiments and why they were chosen.

The naming scheme for the experiments is as following; a *number* describing which model or combination of models is used, and a *letter* uniquely describing experiment. The data of the experiments can be found through the following link.[1] The order in which the experiments are set is illustrated in the road map in Figure 7.1.

[1] https://drive.google.com/folderview?id=0B1mLIuFDRGGERzE0eFlFaUo2N2susp=sharing

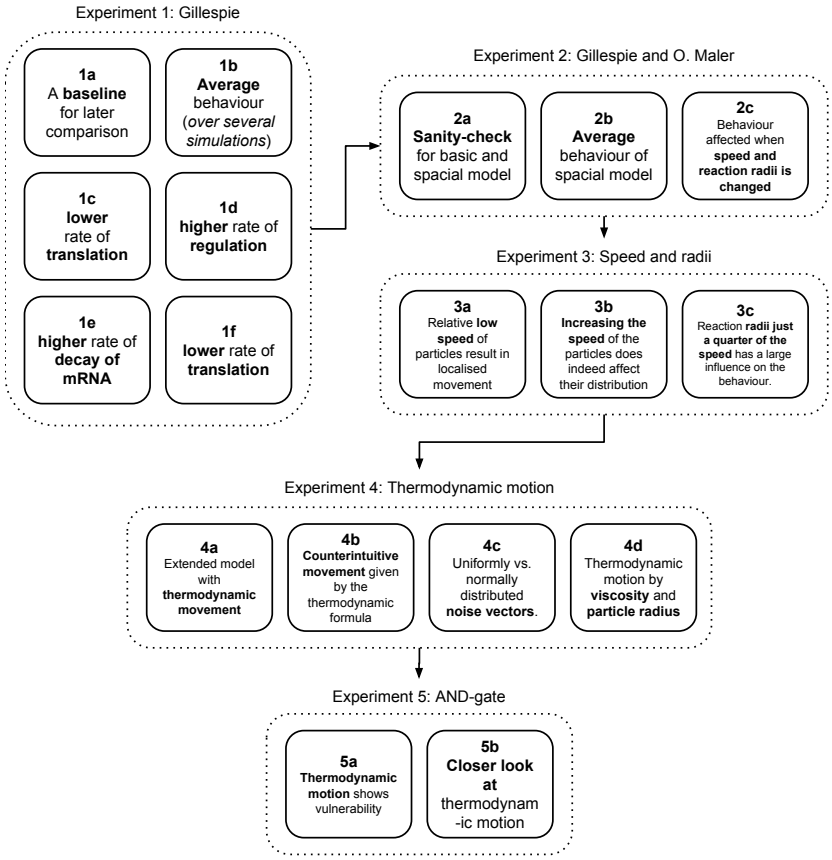


Figure 7.1: A road map showing the experiments conducted in this chapter and the order they are in. The experiment are divided into sub-experiments, where different stages of the models are tested with different parameters in order to confirm or disprove the expected results.

7.1 Experiments

Reaction rates for feedback device:

Below are the reaction rates of the reaction system given by the negative feedback- and and-gate device. These rates describe the behaviour of the device depending on the simulation algorithm i.e. Gillespie. The rates are based on the law of mass actions system i.e. a rate function for each reaction in the system. The

name of the rate function is given followed by the function itself dependent on a rate constant and an amount of a given species. The parameters are based on used in [LB14], in which this device is a sub component of an oscillator device.

`Transcription`	$0.5 * Plac$
`Regulation`	$1 * Plac * LacI$
`Unbinds`	$9 * Plac\_LacI$
`Repressed_transcription`	$5 * 10^{-4} * Plac\_LacI$
`Translation`	$0.167 * mRNA$
`Decay of mRNA`	$0.0058 * mRNA$
`Decay of LacI`	$0.0012 * LacI$

Table 7.1: Negative feedback device parameters

`Promoter activation`	$0.0001 * IPTG * LacI$
`Transcription`	$0.012 * Pro$
`Translation`	$0.009 * mRNA\_Ara$
`Decay of mRNA_Ara`	$0.01 * mRNA\_Ara$
`Decay of Ara`	$0.01 * Ara$

Table 7.2: And-gate device parameters

Experiment 1a

The first experiment will serve as a basis of comparison for the rest of the experiments which are done.

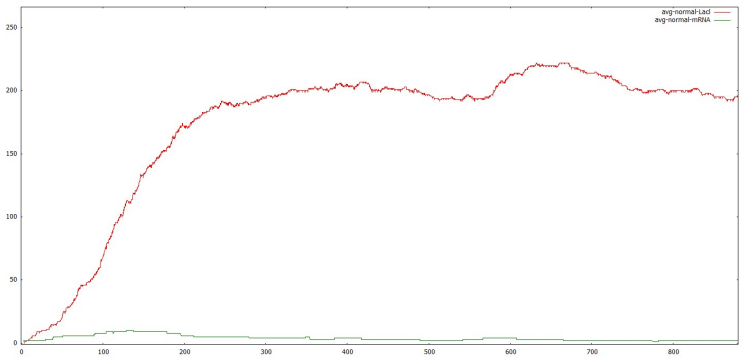
Description	A simulation of the negative feedback device in Figure 3.3, showing the concentrations of mRNA and LacI species over time.
Purpose	To create a baseline of comparison when we refine the model or change parameters or amount of simulations.
Expected result	The concentration of the protein LacI is expected to be repressed at certain point since the chance of regulation happening is proportional to the amount of LacI. The concentration of mRNA is expected to slowly decay over time, since the chance of transcription happening decreases as the frequency regulation increases.
Parameters	See Table 7.1 for device parameters. 1 simulation.
Result	 Run time: $\sim$ 1.5 seconds
Evaluation	The results show that the protein LacI (red) is repressed once it reaches a concentration of $\sim$ 200 particles, although the stochastic element of the simulation presented by the fluctuations of said concentration after repression. One can also see that the mRNA (green) decays over time.

Table 7.3: Experiment 1a, the first axes in the graph is time and second is concentration.

Rational

Experiment 1a in Table 7.3 shows the expected behaviour of the negative feedback device, but in theory it would be hard to determine a confident concentration measurement of the different types of species, based solely on one simulation, because of the stochastic aspects of the system. One way to achieve this, would be to calculate a statistical average over several simulations. The next experiment is to then see if the averaged simulations would still reflect the expected behaviour of the device.

Experiment 1b

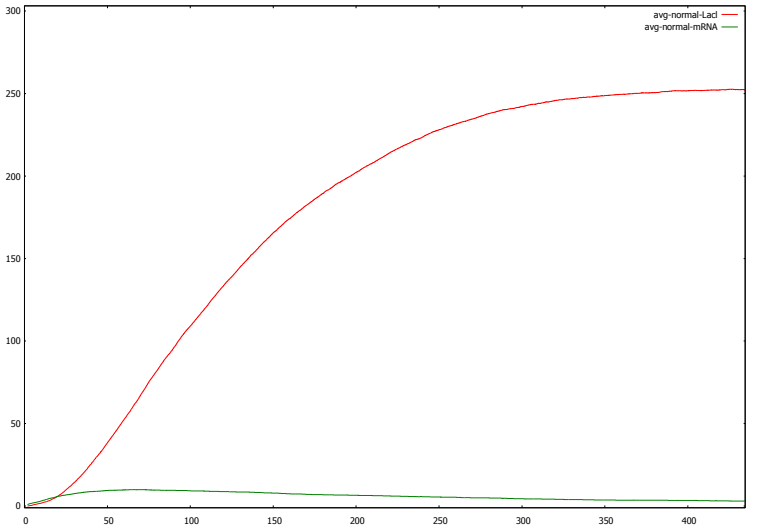
Description	100 simulations of the negative feedback device in Table 7.1, showing the concentrations of mRNA and LacI species over time.
Purpose	To see if the average behaviour of the device is that of the expected.
Expected result	As in experiment 1a in Table 7.3 the concentration of the protein LacI is expected to be repressed at certain point depending on the concentration, after which the amount of LacI is maintained. The concentration of mRNA is also expected to decay over time.
Parameters	See 7.1 for device parameters. 100 simulations
Result	 Run time: ~ 7.8 seconds confidence interval mean LacI at steady-state = $[240.78; 264.78]$
Evaluation	The results show a smooth curve of both the LacI (red) and mRNA (green). We see that in experiment 1a 7.3 the repressed concentration of LacI did in fact deviates from the average by an amount of ~ 50 particles, which illustrates the high variance of the behaviour of the device.

Table 7.4: Experiment 1b

Rational

These first two experiments conclude the 'basic' model used by Gillespie. The results showed the expected behaviour of the device - i.e. repression of specific protein concentration at a given state of the device. The stochastic the behaviour is also illustrated by the high variance and fluctuations of the protein, which are highly dependant on the input parameters used in Table 7.1 which is also concluded in [LB14]. So the next four experiments are based on testing the hypothesis that these parameters are indeed affecting the overall behaviour of the device.

Experiment 1c

The following experiments will test the device under different parameters in terms of reactions rates of translation, regulation, decay of mRNA, and translation. The experiments are conducted on the hypothesis that the behaviour of the devices is closely affected by the inputed parameters of the user.

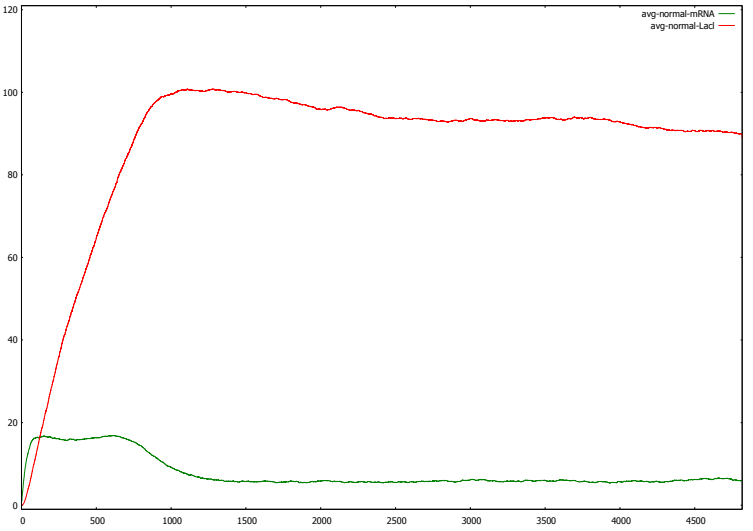
Description	100 simulations of the negative feedback device in Table 7.1, showing the concentrations of mRNA and LacI species over time. The rate of translation is lowered by a factor of 10^{-1}.
Purpose	To see if the average behaviour of the negative feedback device is affected by the rate of which translation happens, i.e. when mRNA is translated into the LacI protein.
Expected result	It is expected that by lowering the translation rate, the amount of LacI presented in the cell should be lower compared to the results in Table 7.4.
Parameters	See 7.1 for device parameters, with the rate of translation lowered by a factor of 10^{-1}. 100 simulations, duration = 10x of that in Table 7.4.
Result	 Run time: $\sim$ 71 seconds
Evaluation	The steady state of LacI rests at a concentration of $\sim$ 100, in which the concentration of mRNA is noticeably higher at the start of the simulation. The much larger x-axis indicates that reactions where mRNA or LacI are produced are chosen not as often. This shows that the time for the device to reach its steady state is much higher.

Table 7.5: Experiment 1c

Rational

We see that by lowering the rate of translation we do still achieve the desired behaviour of the LacI protein reaching a steady state. Though it should be noted, by the lower concentration, the lifespan of the device itself would be shorter given by the proportionally faster decay of the protein.

Experiment 1d

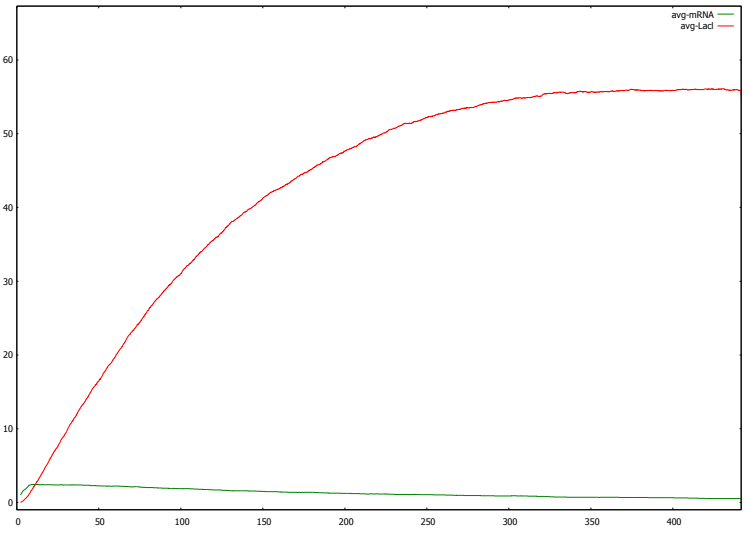
Description	100 simulations of the negative feedback device in Table 7.1, showing the concentrations of mRNA and LacI species over time. The rate of regulation is increased by a factor of 100.
Purpose	To see if the average behaviour of the negative feedback device is affected by the rate of which regulation happens, i.e. when a LacI protein meets the promoter and produces PlacLacI.
Expected result	By increasing the rate of regulation, we specify that LacI proteins meet the promoter more often, in which is is consumed to produce another protein. By this, we can expect a lower steady state concentration of LacI.
Parameters	See 7.1 for device parameters, with the rate of translation increased by a factor of 100. 100 simulations
Result	 Run time: $\sim$ 30 seconds
Evaluation	The steady state of LacI rests at a concentration of $\sim$ 55. This is most likely caused by the expected effect of increasing the rate of regulation - that more LacI proteins are the reactants for said reaction.

Table 7.6: Experiment 1d

Rational

The results indicate that regulation solely maintains the concentration level of LacI, and does not affect any of the other aspects of the dynamic behaviour of the device.

Experiment 1e

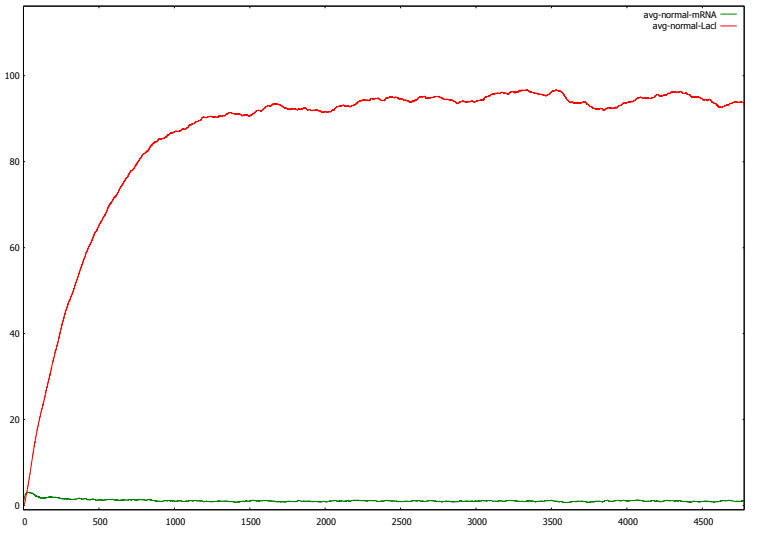
Description	100 simulations of the negative feedback device in Table 7.1, showing the concentrations of mRNA and LacI species over time. The rate of decay of mRNA is increased by a factor of 10.
Purpose	To see if the average behaviour of the negative feedback device is affected by the rate of which decay of mRNA happens.
Expected result	If the transcribed mRNA strands decay much faster, the amount of LacI translated should thus be affected in some way.
Parameters	See 7.1 for device parameters, with the rate of decay of mRNA increased by a factor of 10. 100 simulations, duration = 10x of that in Table 7.4.
Result	 Run time: $\sim$ 200 seconds
Evaluation	We see that the amount of mRNA present in the cell indeed has an effect on the concentration of LacI. Again, as seen in Table 7.5, the time it takes for the device to reach its steady state is much higher.

Table 7.7: Experiment 1e

Rational

We see that by decreasing the rate of which decay of mRNA happens, that regulation is not the sole factor that describes the steaty-state of LacI.

Experiment 1f

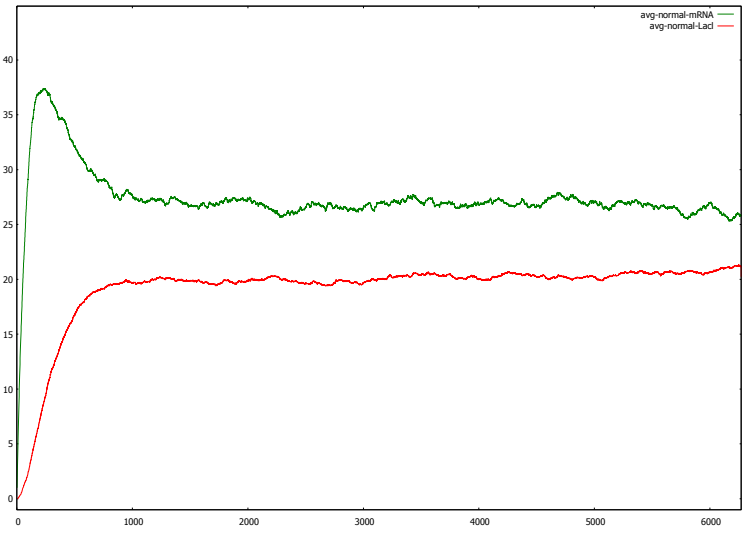
Description	100 simulations of the negative feedback device in Table 3.3, showing the concentrations of mRNA and LacI species over time.
Purpose	To see if the rate of translation has any further effect by lowering it even more compared to that in Table 7.5 by a factor of 10^{-2}.
Expected result	If we lower the rate of translation even more, we should see an increase of mRNA, since it will not be translated into the LacI protein as often. Which will also result in a lower concentration of LacI.
Parameters	See 7.1 for device parameters, with the rate of translation lowered by a factor of 10^{-2}. 100 simulations, duration = 10x of that in Table 7.4.
Result	 Run time: $\sim$ 194 seconds
Evaluation	We see that, as the mRNA reaches a concentration of $\sim$ 38, translation is the most likeliest to be chosen by the algorithm, thus resulting in the consumption of mRNA and lower steady-state of $\sim$ 27.

Table 7.8: Experiment 1f

Rational

This concludes the experiments for the basic model, not taking the spatial model into consideration. We saw that, given by the parameters in Table 7.1, the mean concentration of LacI in its steady-state would lay in the interval of $[240.78; 264.78]$ with a 95% level of confidence described in Chapter 4. This will later be our point of comparison, when we do a sanity-check of the spatial model developed, and later experimentation of thermodynamic model of motion. The last four experiments showed the vulnerability of the device, which relates back to the importance of the parameters we use in general when simulating synthetic genetic devices.

Experiment 2a

The following experiments will compare the basic model with a refined one taking the particles exact position within the cell into account in order to simulate collisions, compared to Gillespie's that estimates collisions in a statistical manner, this model is an adaptation by the model proposed in [MHML14] and is referred to as the 'spatial' model.

The size N describes a N^3 square cell, the radius r describing the reaction radii for all particles, the temperature T in Kelvin, the motion m either defined as a constant or a function describing the velocity of the particles i.e. be it constant or a function of temperature etc., the elasticity constant e describing the energy loss of cell wall interaction.

The size of the cells are set to reflect the size of a virus cell, which varies, but mostly is about 100-120 nm. On the other hand, bacterial cells are quite larger by a magnitude of 10 compared to the virus cells, the smallest being a *Escherichia coli* cell [Kub90] being 2 μm long and 0.5 μm in diameter. But for the purpose of clarity, the size of a virus cell is used as the basis for N. Later N will be tested once we have evaluated the motion of particles.

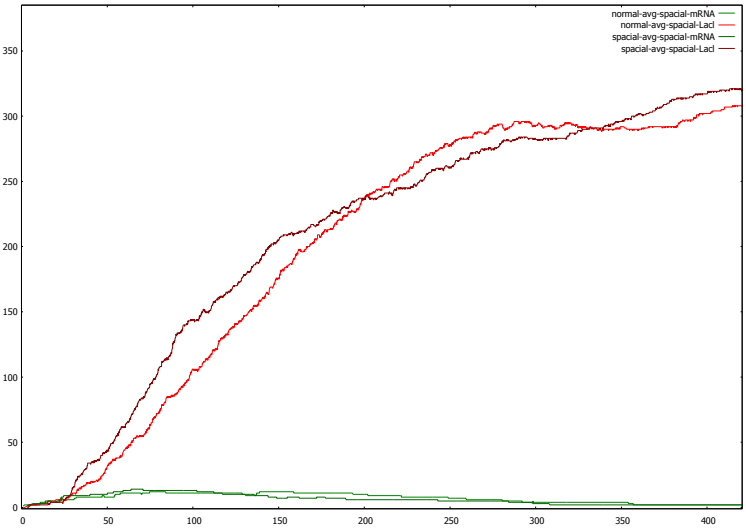
Description	Simulations comparing the basic model with the refined version taking the particles exact positions into account. In this experiment the particles do not move hence the purpose described below.
Purpose	This experiment should serve as sanity-check, i.e. if the new model illustrates the same behaviour under parameters describing the basic model - the speed of the particles is set to 0 i.e. not moving thus allowing particles to react with each other no matter their relative positions.
Expected result	We expect that the concentrations of both mRNA and LacI for both models to be close to equal. But due to the stochastic behaviour of the device we should not expect exact equal results, but one should at least be able to see some correlation between the two result sets.
Parameters	For the spacial model: $N = 100.0$; $r = 129.0$; $T = 1.0$; $m =$ `Constant(0.0)`; $e = 0.5$; 1 simulation It should be noted that once the motion is defined as '`Constant(0.0)`' the particles will not move, hence reflecting a device where all the cells are close enough to each other to react. The radius is also set to be arbitrarily larger than the cell size to ensure the same. The temperature and elasticity can be ignored in this case.
Result	 Run time: ~ 1.3 seconds
Evaluation	We see that the LacI concentrations in the spacial model (dark red) follows the basic model (red) in some sense, but again, due to the fluctuations it remains inconclusive whether the two results sets reflect the same behaviour.

Table 7.9: Experiment 2a

Rational

When comparing the two result sets of just one simulation of each model, the resemblance between them remains inconclusive. This, again, motivates an averaged behaviour of several simulations.

Experiment 2b

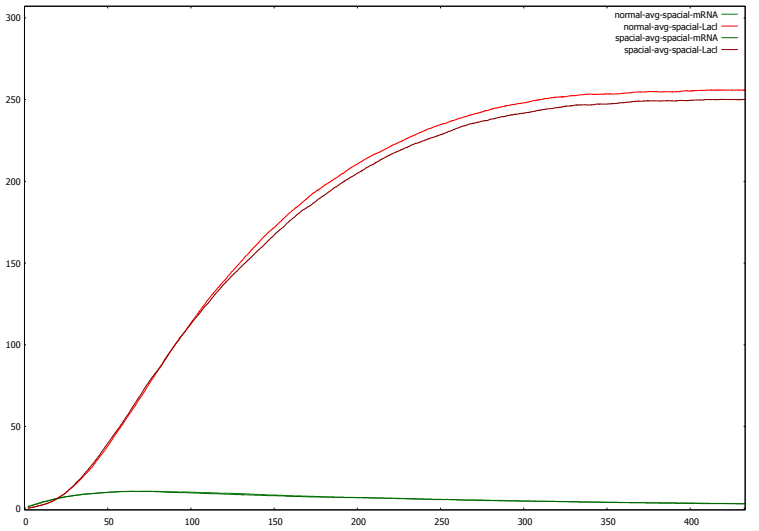
Description	100 simulations to test if we can see a correlation of the two models with the same input parameters as in 7.9.
Purpose	This experiment should show that the spacial model reflects the same behaviour as the basic model under a specific set of parameters, as mentioned in Table 7.9, but by comparing the average behaviour based on an increased amount of simulations.
Expected result	We expect that the concentrations of both mRNA and LacI are close to equal for both models. When increasing the amount of simulations, one should see a close-to-overlapping lines for LacI and mRNA concentrations.
Parameters	The parameters remain as in 7.9, but with simulations set to 100 for both models.
Result	 Run time: $\sim$ 96 seconds
Evaluation	When increasing the amount of simulations the concentrations of LacI in both of the models get closer to each other, they are not exact overlapping, but this experiment shows that the average behaviour is indeed the same for both model under said parameters, since 250 lays within the interval we found earlier.

Table 7.10: Experiment 2b

Rational

The average concentrations of LacI are not exactly overlapping, but achieving this would potentially require are much larger amount of simulations. Thus the sanity check for the spacial model is concluded. The next step is to then see if the positions of the particles in fact matter for the device.

Experiment 2c

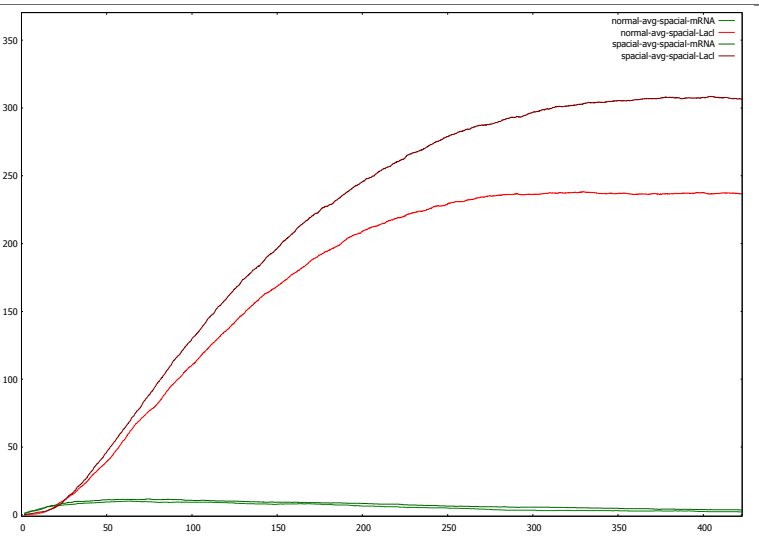
Description	Simulation that will compare the basic model with the spacial model used in 7.9 and 7.10 but with altered parameters for the spacial model reflecting moving particles.
Purpose	The purpose of this experiment is to show that increasing the speed and decreasing the reaction radii of the particles in the cell influences the behaviour of the device in some way.
Expected result	With an increased speed and decreased reaction radii compared to the 'stationary' particles in Table 7.9 and 7.10, we should see a decreased repression rate of LacI, given the intuition that the rate of regulation in the device will decrease since more LacI particles will not be close enough to react with the Plac promoter.
Parameters	The parameters for the basic model remain. For the spacial model:space = $N = 100.0$; $r = 1.0$; $T = 1.0$; $m =$ `Constant(0.50)`; $e = 0.5$; 100 simulation
Result	 Run time: $\sim$ 78 seconds.
Evaluation	We clearly see, compared to Table 7.10, that the repression rate is indeed decreased thus allowing a larger amount of LacI to be present compared to the basic model, which reflects a more uniform distribution of slowly moving particles, i.e. having a small chance of returning to their origin.

Table 7.11: Experiment 2c

Rational

This concludes the experiments comparing the basic model with the spacial model. We saw that increasing the speed by a small amount relative to the cell size, had an influence on the behaviour of the device. The experiment in 7.11 also shows that the device could potentially be 'broken' if the distribution of the particles get closer to be uniform, given the particles have a low enough speed such that they stay distant from each other. But the exact motion and distribution are hidden in said experiments, which motivates animations of the device in 3D enabling more in depth analysis of the device.

Experiment 3a

The following experiments will test the spacial model on the hypothesis that the speed of the particles relative to the size of the cell has an influence on how they are distributed, and if so, it should affect the behaviour of the device. The movement is modelled such that particles move freely throughout the cell without interacting with each other, and when they bump into the barrier/membrane of the cell, they will simply bounce back losing some kinetic energy in the process based on the *elasticity* parameter. Their movement is modelled to reflect Brownian motion as a noise vector with mean $\mu = 0.0$ and variance $\sigma = 1.0$ in a uniform distribution.

In order to evaluate the behaviour in terms of movement of the particles, animations of 3D scatterplots have been rendered for all relevant experiments and can be accessed through the following link[2]. The naming scheme for the experiments is adapted for the videos e.g. the video for experiment 3a is called '`EXP3A`'.

[2]https://drive.google.com/folderview?id=0B1mLIuFDRGGERzE0eFlFaUo2N2susp=sharing

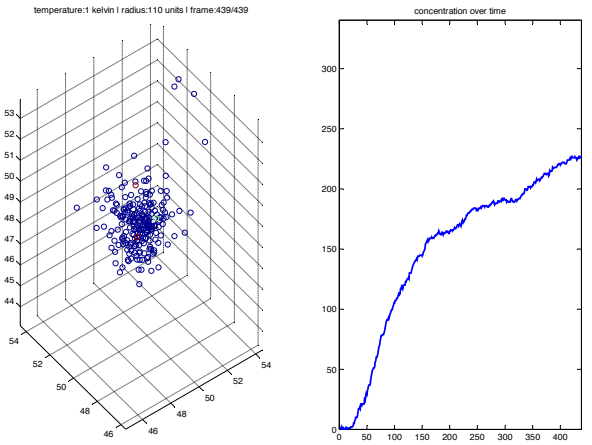
Description	A simulation showing small and localised movement of particles. The result is presented in a 3D scatter plot on the left and a graph similar to the previous experiments on the right showing the concentration of the LacI protein over time.
Purpose	The purpose of this experiment is to test if a relative low speed of the particles will result in particles moving in a localised area around the device, given by the start position the promoter.
Expected result	The particles are expected to move around in a small area relative to the size of the cell. Given the following nature of the device: starting with just one Plac promoter that transcribes mRNA, that is translated into the LacI protein, we can predict that the LacI particles will be closely distributed around the promoter.
Parameters	The parameters are the same as in Table 7.11 with $r = 110.0$; 1 simulation.
Result	 Run time: ~ 0.74 seconds.
Evaluation	We see that the particles indeed move in a localised area (LacI protein (blue), mRNA (red), and Plac(green), and we maintain the behaviour illustrated in the graph to the right. The behaviour is only maintained since we have set the reaction radius to larger than the cell itself - i.e all particles can reach each other.

Table 7.12: Experiment 3a

Rational

The reason behind allowing all particles to react with each other no matter the distance between them, was to maintain the behaviour while moving the particles. The localised movement shown was achieved with a low speed, which motivates the next experiment with increased speed.

Experiment 3b

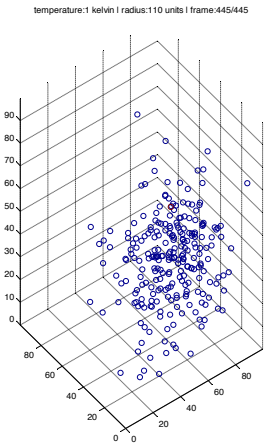
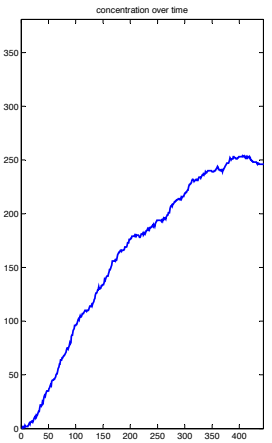
Description	A simulation with the same parameters as in Table 7.12, but with an increased particle speed.
Purpose	This experiment should show that increasing the speed of the particles does indeed affect their distribution, which would then affect the behaviour of the device as seen in Table 7.11.
Expected result	We expect the distribution of particles to be closer to uniform relative to the entire cell caused by the increased speed.
Parameters	The parameters are the same as in 7.12, but with motion set to $m =$ `Constant(10.0)`; 1 simulation
Result	  Run time: ~ 0.85 seconds.
Evaluation	The scatter splot shows the last recorded state of the simulation. Compared to the results in 7.12, wee see that the cells are indeed more uniformly distributed throughout the cell.

Table 7.13: Experiment 3b

Rational

Now that we have shown that the speed of the particles influence the distribution, it now makes sense to see if it also affects the behaviour of the device, thus by decreasing the reaction radius to be more 'sensible' compared to the unrealistic length, greater than the cell itself, used in the experiments above.

Experiment 3c

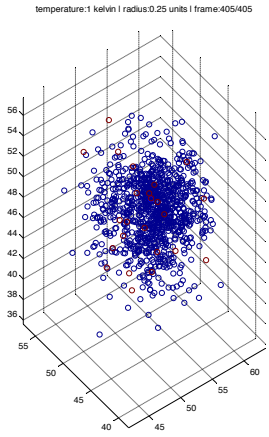
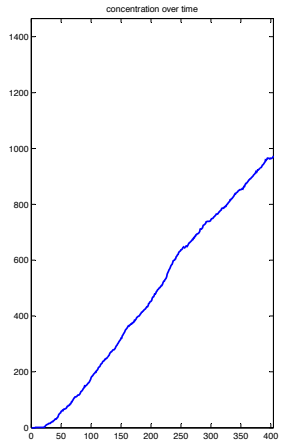
Description	This experiment will take more 'sensible' parameters into account, one being the ratio between the reaction radii and the speed of the particles.
Purpose	The purpose of this experiment is to show that having a reaction radius (size of the particles) just a quarter of the speed has a large influence on the behaviour of the device.
Expected result	It is expected that we will see an increased amount of LacI compared to the last two experiments, since fewer Plac and LacI particle will be neighbouring each other for the regulation reaction to happen, resulting in lower repression rate of LacI.
Parameters	$N = 100.0$; $r = 0.25$; $T = 1.0$; $m =$ `Constant(1.0)`; $e = 0.5$; 1 simulation
Result	  Run time: ~ 5 seconds.
Evaluation	Even though the particles stay localised in the same area in the cell, the much lower reaction radius has had a major impact on the behaviour of the device resulting in an ever increasing concentration or indeterminable steady-sate of the LacI protein (blue), which in this case means that we have 'broken' the device i.e. the expected behaviour is lost.

Table 7.14: Experiment 3c

Rational

The experiments in this section prove the said hypothesis - that the speed of the particles affects their distribution relative to each other, and thus also affects the behaviour. But, as mentioned earlier, the way we model the movement of the particles does not take collisions of particles and other physical aspects, such as *hydrodynamic interactions*, into account. This motivates further refinements and experiments of the spacial model.

Experiment 4a

The following experiments will test refinements done to the spacial model in terms of how particles move in the cell. This will be done by gradually extending the 'motion' parameter to be more true-to-nature and realistic by adding additional parameters:

- Boltzmann constant is set to $1.3806488 * 10^{-23}\ J/K$ [MTN11].
- The cell sizes are found in [Lea]. Where viral cells are about 100 nm, and its containing particles to be about 4 nm.
- Determining the weight of the particles, is done on the basis of a LacI protein weighing 38.59 kD (kiloDalton) [Bio], which is about $6.40 * 10^{-20}$ grams.
- The temperature of a cell is expected to be at room temperature i.e. 298 Kelvin.
- The viscosity of cytoplasm is said to be 8 times greater than water [MBTK84] [IMR86] (which has a viscosity of 1).

The following experiments will use thermodynamics in order to determine the velocity of the particles depending on the temperature within the cell. The velocities are determined by the formula described by Equation 3.6 in Chapter 3.

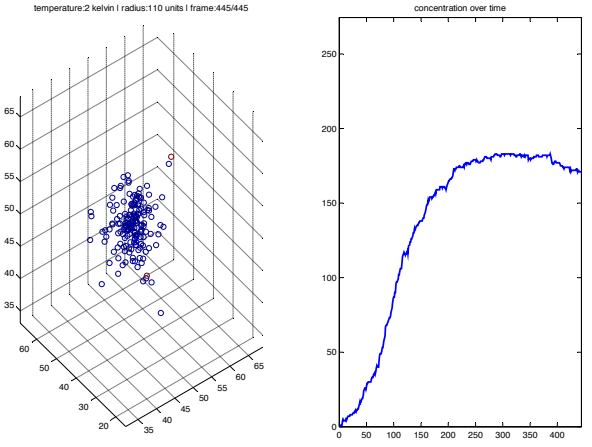
Description	A simulation with the thermodynamic model described above, with parameters that result in the same motion behaviour as seen earlier.
Purpose	This experiment should show that the model can easily be extended in terms of motion, in this case a formula determining the velocity of the particles as a function of temperature.
Expected result	The distribution of particles is expected to be similar to what was found in Table 7.12.
Parameters	$N = 100.0$; $r = 110.0$; $T = 2.0$; $m =$ `VelocityAtTemp(fun temp -> sqrt((3.0*1.3806488e-23*temp)/0.0005)*10.0e+9)`; $e = 0.5$; 1 simulation
Result	 Run time: ~ 1.46 seconds.
Evaluation	We see that we need a temperature of just 2.0 Kelvin and a particle mass of 0.0005 g, which is quite unrealistic, in order to maintain the behaviour of the device.

Table 7.15: Experiment 4a

Rational

The behaviour achieved by the rather unrealistic parameters shows the abstractions of the current model, one of them being the abstraction of non-colliding particles but also the fact that the thermodynamic formula models particles moving in a vacuum. But the localised movement of particles is based on the intuition that they do not move in a rapid fashion within the entire cell, which also constitutes to the rather unrealistic parameters needed in order to achieve the desired behaviour. But what if the particles do in fact move rapidly throughout the entire cell? To illustrate this, in the next experiment we try to input some realistic parameters for the cell size, protein size (reaction radius), and temperature.

Experiment 4b

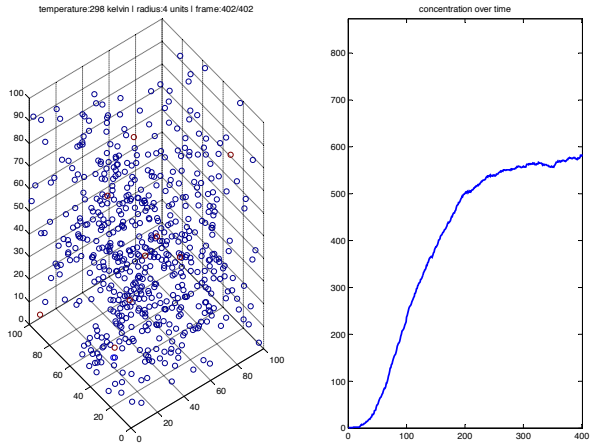
Description	This experiment will show the behaviour under realistic parameters when the model is refined in terms of particle movement by the thermodynamic formula described earlier.
Purpose	The purpose of this experiment is to show the limitations of the current model, as stated in the rational before.
Expected result	Particles are expected to move at high speed, given that we drastically decrease the mass of the particles and raise the temperature in the cell to room temperature i.e. 298 degrees Kelvin.
Parameters	$N = 100.0$; $r = 4.0$; $T = 298.0$; $m =$ `VelocityAtTemp(fun temp -> (sqrt((3.0*1.3806488e-23*temp)/6.4e-22))*10.0e+9)`; $e = 0.5$; 1 simulation
Result	 Run time: $\sim$ 13.35 seconds.
Evaluation	An interesting observation to make here, is that with the uniform distribution and rapidly moving particles, the model still reflects the expected behaviour in this environment, though with a lower repression rate resulting in much higher steady-state concentration of LacI.

Table 7.16: Experiment 4b

Rational

The last two experiments motivates further refinement of the model, such as computing the force from interaction with neighbour particles etc.. The noise vector generated in order to achieve the Brownian motion, is as mentioned of uniform distribution. But what if it was normally distributed i.e. in the nature of *Gaussian noise* as described in [AS13]? The next experiment thus compares a uniform noise voice against a Gaussian noise vector.

Experiment 4c

In order to specify what kind of noise utilised when moving particles, the spacial model is extended with a distribution parameter d. The normal distribution is of mean $\mu = 0.0$ and variance $\sigma = 0.25$, describing a noise vector in the range of $([-1; 1], [-1; 1], [-1; 1])$ - the same as the uniformly distributed noise vector used in previous experiments.

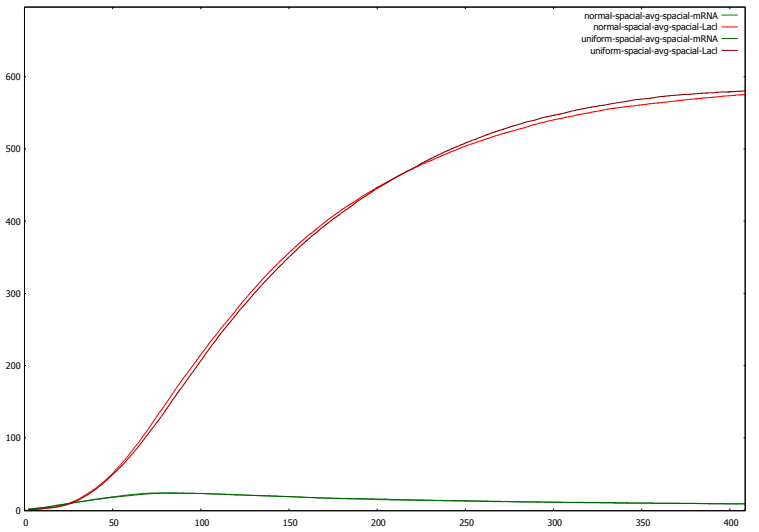
Description	100 simulations for two different kinds of noise vectors describing the Brownian motion.
Purpose	The purpose of this experiment is to see if the noise vector has influence on the movement of the particle thus the behaviour.
Expected result	When the movement is normally distributed instead of uniformly, the intuition here is that the particles will move in lower average speed. The lower speed will result in a lower chance of regulation of the promoter and a LacI protein.
Parameters	The parameters for both of the models are the same as in 7.16, but with extra parameter $d =$ `NormalDist(normalDist)` for the normal distribution simulation and $d =$ `UniformDist(uniformDist)` for the uniform distribution simulation. 100 simulations for each model. Both of the number generators are from the `MathNet.Numerics.Distributions` library, where `uniformDist` is an instance of `ContinuousUniform` and `normalDist` is of `Normal(mean, stddev)` with the respective mean and standard deviation described before.
Result	 Run time: $\sim$ 14 minutes 18 seconds confidence interval mean LacI at steady-state $= [543.21; 582.47]$ of `uniformDist`.
Evaluation	We see that changing the motion to be normally distributed has little to no affect on the repression rate of LacI, which confirms the intuition that though the average movement will be lower compared to the uniformly distributed motion, the relatively high velocity of the particles as still high enough to achieve the same behaviour when the noise vector is normally distributed.

Table 7.17: Experiment 4b

Rational

Given the results found in Table 7.16 the thermodynamic formula does not provide a suitable environment for testing different noise vectors. This motivates the introduction of a refined formula that takes the viscosity and size of the particles into account in terms of their movement.

Experiment 4d

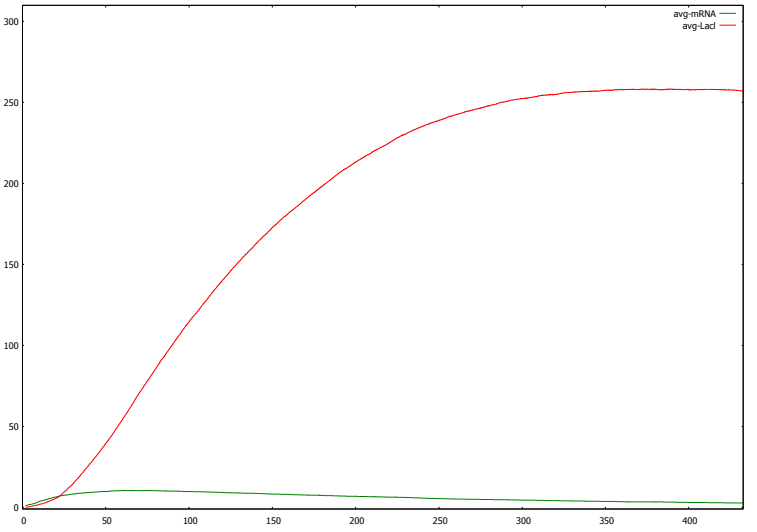
Description	100 simulations of the negative feedback device, in which the thermodynamic formula described by Equation 3.7 is utilised.
Purpose	To see if the refined model for thermodynamic motion is sufficient to simulate the expected dynamic behaviour of the negative feedback device.
Expected result	We now know that the speed relative to the size of the particle is what determines of the device works or not. Given by Equation 3.7, we introduce drag proportional to the size of the cell which will decrease the speed. But it is hard to predict if it decrease enough.
Parameters	The parameters are the same as in Table 7.17, but with the motion replaced with `VelocityAtTempRad(fun temp rad -> sqrt(2.0*kB*temp/(3.0*pi*eta*(alpha rad))*unitScale))` where `alpha` is a function describing the drag proportional to the radius of the particles, and `eta` the viscosity set to 8. 100 simulations.
Result	 Run time: $\sim$ 99 seconds
Evaluation	We clearly see that this model of thermodynamic motion might be sufficient to describe the motion of particles within a cell, given the parameters used and device that is simulated. Compared to the result in Table 7.17, we did reach a steady-state within confidence interval.

Table 7.18: Experiment 4d

Rational

This concludes the experiments of the thermodynamic models of motions introduced in this thesis and comparison of noise vectors. As the results show, there is room for refining the model with hydrodynamic interactions, which could lead to lower speeds [AS13]. In the following last set of experiments, we will try and see if another device, i.e. the and-gate device illustrated in Figure 3.4 in Chapter 3, works under the same conditions.

Experiment 5a

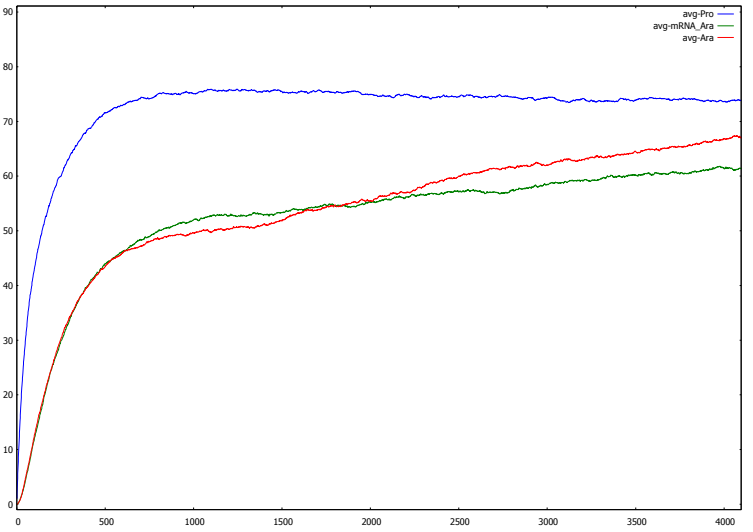
Description	100 simulations of the and-gate using the basic model.
Purpose	To later compare the basic model with the spatial where the cell contains 100 of each proteins that have to meet in order to activate a steady-state.
Expected result	We expect the IPTG and LacI proteins to meet and activate the promoter, such that mRNA is transcribed and translated into mRNA_Ara.
Parameters	The reaction rates of the chemical reaction system describing this device are listed in Table 7.2. The important aspect of these experiments, is that two proteins have to meet in order to produce a promoting protein that produces a steady-state of a protein. Concentration of IPTG and LacI is set to 100. 100 simulations.
Result	 Run time: $\sim$ 10 seconds. confidence interval mean Pro at steady-state = $[73.74; 76.29]$
Evaluation	We see that the basic model simulates the expected behaviour, i.e. a steady state of mRNA_Ara protein.

Table 7.19: Experiment 5a

Rational

This experiment illustrates the expected behaviour of the and-gate device, showing a steady-state of a given protein activated by two other proteins. The core motivation behind choosing this device for testing, is that we intuitively can say if the key proteins do not meet often enough the behaviour of the and-gate will be different, when we simulate it with the spatial model. Although, given by the relative high initial amount of particles, we could predict that the device would not be effected by this.

Experiment 5b

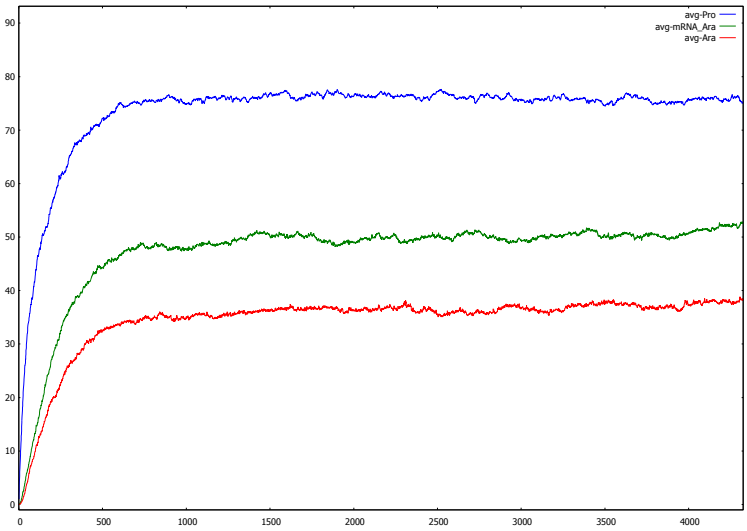
Description	10 simulations of the and-gate using spatial model with the same parameters as in Table 7.19.
Purpose	To see spatial model will simulate the same behaviour of the and-gate as in the basic model.
Expected result	It is hard to predict, whether the behaviour is maintained or not, when we take the positions of the particles into account. One could, as mentioned, intuitively say - given by the law of mass action described in Chapter 3, the high concentrations would lead to a sufficient propensity function for the given reaction needed for activating the steady-state.
Parameters	The parameters are the same as in Table 7.18. 100 simulations.
Result	 Run time: $\sim$ 6 minutes.
Evaluation	We see that we have the same concentration of the Pro protein, but different concentrations of Ara and mRNA_Ara. This could be due to the reaction producing mRNA might be chosen at a lower rate, since the reaction producing Pro is chosen more often even though there are no neighbouring pairs to be found.

Table 7.20: Experiment 5b

Rational

The behaviour of the and-gate is maintained when utilising the same thermodynamic model as in Table 7.18. Though whether the concentrations of mRNA or mRNA_Ara is of importance is unclear, but if so, the behaviour is not maintained fully. We see that by having a population size of 100 particles of each protein, as seen in Appendix B, we have maintain the propensity function for the relating reaction. As mentioned in Chapter 3, this was an interesting point of experimentation; to see if the rate of reaction is in fact proportional to the population size of the given species.

7.2 Summary

The purpose of this chapter was conduct experiments, testing the negative feedback- and and-gate device under different conditions. A main conclusion can be made in terms of the vulnerabilities of synthetic genetic device:

The speed of which the given particles of the negative feedback device move, compared to their size, is of great importance.

This was found in experiment 3c in Table 7.14, showing that a particle size just a quarter of the speed had a great impact on the behaviour.
Later we saw that introducing the thermodynamic model of motion described by Equation 3.6 in Chapter 3, required unrealistic parameters in order to maintain the behaviour of the negative feedback device. From this, we can conclude that:

The thermodynamic model of motion used has a major impact on the speed of the particles, thus affecting the behaviour of the device.

When we then tried to input realistic parameters and utilise the model in Equation 3.7, we were able to maintain the behaviour. But, it is still motivated to introduce a refined model concerning hydrodynamic interactions.

The last experiments, where we simulated the and-gate device, showed that the well-stirred assumption made by Gillespie might be sufficient for this specific device.

The parameters used are still a rough estimate. One being that the species/-particles are not the same size and weight and, for that reason, do not at the same speed. E.g. considering the produced protein to be the same size and

structure (spheres) as a strand of mRNA, is a crucial assumption that is made by the utilised spatial model. This assumption, among others, motivates further refinements to the spatial model, such as considering mRNA strands as complex structures.

Chapter 8

Conclusion

In this thesis we have shown how to model and analyse synthetic genetic devices with focus on particle collisions, to see if the current means of simulating such suffice.

8.1 Summary

As stated in the introduction in Chapter 1, the proposed problem required the implementation of a suitable framework for simulating synthetic genetic devices, that should be easily extensible and modifiable.
This was done by stating some minimum requirements in Chapter 2, in short describing a tool for simulation any kind of chemical reaction system specified in SBML format. Before specifying the structure of the design in Chapter 4, research had to be done in terms of how synthetic genetic devices can be simulated whilst assessing the level of abstraction of the different approaches.

In Chapter 3 two models were then compared - Gillespie's stochastic model and Oded Maler's spatial model. The key difference between the two was evaluated in terms of describing particle collisions, where Gillespie's model includes a statistical estimation compared to Oded Maler's, describing actual particle collisions. A combination of the two models was then proposed in which Gillespie's

direct method was extended with Oded Maler's approach of finding and choosing neighbouring particles for reaction, thus maintaining stochastic behaviour. Lastly, different thermodynamic models of motion were introduced extending the model to be as true-to-nature as possible, e.g. simulating the devices under realistic parameters, such as the temperature within the cell.

The design of the tool was then introduced by stating the different modules in terms of their role and functionality relative to the whole system. The functional paradigm provided us with easy model extension, e.g. when different data structures were needed for simulation particle motion and interaction. Describing a polymorphic type instantiated with specific parameters and functions then provided us with the flexibility needed for e.g. comparison and extensibility. The level of cohesion between the different modules was kept as low as possible, later enabling a simulator module taking any kind of simulation algorithm and data structure describing the chemical reaction system.

Performance was a concern during implementation, and the process of measuring such and deploying optimisations was a key exercise. The achieved performance gains were done with little effort in terms of lines of code needed by parallelising the simulator and neighbour search, after which a speed up relative to the number of cores available was achieved. Given the run times listed in Chapter 7, ranging from a couple of seconds to several minutes, the performance highly depends on the parameters specified for the device, indicating the bottleneck when finding neighbouring particles.

As mentioned in Chapter 6 the tests for the tool were done during implementation, leading to little documentation of such. Though given by the extensive experiments done in Chapter 7, an evaluation of the functional and non-functional requirements can be proposed through these.

8.2 Evaluation

The central question that arose during the work of this thesis, concerning a sufficient level of abstraction of the models were:

Does a statistical estimation of particle collision sufficiently describe how particles interact within a synthetic genetic device?

As concluded by the experiment conducted in Chapter 7, we can answer this by the following:

Given realistic parameters for temperature and sizes of protein etc., we can say that Gillespie's model of statistical estimation *is in fact sufficient*. This is mainly a result of the low net movement of the particles when utilising a thermodynamic model of motion. This caused the particles to be situated in a localised area around the origin of the chemical system the device is described by. A key observation made during the experiments, was that the speed of the particles relative to their reaction radii, described how the particle were progressively going to be distributed in the given environment. And as the speed of the particles increased (though to unrealistic heights), the negative feedback device was close to working.

When looking back at the functional requirements stated in Chapter 2, we can evaluate these as follows:

- The parser was implemented such that it can parse SBML files of a specific structure exemplified in Appendix A. The parser is then not complete in terms of understanding the whole SBML specification listed in [RM14]. Given the extend of this specification, it was chosen not to put further work into the parser. If one was to later extend the parser, the process of doing so would include extending the lexer, followed by the parser returning the correct sub types of the model described in Appendix C.
- The model, that is later compiled into an SPN, follows the SBML specification fully. Some constructs of the SBML files used for experimentation do defer from the SBML specification, in terms of which parameters are optional or not.
- The simulator module was implemented, such that it can handle any kind of algorithm for simulating chemical reaction system, following the correct type signature as specified in Chapter 4. The signature of the algorithm limits the possible kinds of algorithms, but still defines a generic type of a discrete event simulation algorithm.
- Parametrising a simulation is done by specifying the duration and amount of simulations, amongst others. These are declared as hard values within the given script issuing the simulation. One could argue, it would be suitable to specify these parameters in the given SBML file, enabling a simple tool where SBML is the sole interface. From a usability point of view, the gains of doing so would greatly depend on how the given GUI of an actual tool, utilising this framework, would work.
- Examples of how the data is outputted by the simulator module are shown in the experiments in Chapter 7. A module for presenting simple plots was implemented for easy and quick evaluation. When we later wanted to verify the motion of particle against our stated hypothesises, we needed a

more sophisticated means of visualisation. This was done by implementing a small Matlab script, so that the particles movement and interactions would be revealed. One could have tried to explore what possible solution for such exists within the `.NET` framework. But given the broad and mature tool set for data visualisation provided by Matlab, it only required means of data extraction and a few lines of code in Matlab to enable 3D scatter plots and animations.

- Calculating and evaluating statistical measures of a simulation correctly, was of great importance and contains some common pitfalls. When calculating the 'average' behaviour of a device of a set of simulations, was done by finding a moving average, which differs from the general average. Confirming that two results set reflect the same behaviour, then relied on performing some statistical evaluation provided by a level of significance. One could argue that evaluating the last measurement of a concentration of given species as the basis for the comparison, might not be enough, and a more sophisticated comparison must be made.

As mentioned in the summary, the main non-functional requirements was a modular and extensible framework. The described design was achieved first evaluating the key types needed to describe the flow of which data should be exchanged between the modules, given by the requirements and information stored in the model.

8.3 Future work

During the research and implementation of the tool, many choices had to be made in order to narrow down the scope of this thesis. This lead to exploration of the different branches of particle simulation and computer scientific approaches to analysis of discrete continuous systems. The most essential aspects of these are as follows:

- **Statistical model checking**: Performing statistical model checking as proposed in [NH14] and [DLL+15], would serve as another means of estimating and evaluating the behaviour of a synthetic genetic devices. Doing so, the entire result from a set simulations would be evaluated by stating some logical condition that must be uphold. Given the comparable analogy used in [NH14] to the SPN, this could be achieved by utilising the already implemented features of conducting several simulations, after which this form of evaluation could be applied.

- **Static analysis**: The inputted SBML files might contain inconsistencies or spurious reaction statements, either causing unwanted behaviour or exceeding the technical limitations of the hardware, e.g. reaching a upper limit of particles. This relates directly to the subject of static analysis within computer science, better known a program analysis. As proposed in [NNH99], an over-approximation given by an interval-analysis could be utilised for estimating memory vulnerabilities caused by the given set up.
- **Faster neighbour search**: As emphasized in Chapter 5, finding neighbouring particles is the main bottle neck when ultilising the spatial model. Many aspects of particle motion, size, and diffusion tensor calculation are crucial, which performance relies on the given data structure used. Introducing these optimizations will be important if/when the future genetic devices grow in complexity in terms of acting parts.
- **Interactive GUI/visualisation**: An obviously missing part of the tool, presented in this thesis, is a 'binding' component, combining the components thus providing the functionalities needed for e.g. specifying parameters for simulation and initiating illustrations. When creating such a GUI, it will be important to take note on the work flow of a biochemist etc., introducing some aspects of user experience- and usability engineering.
- **Model refinements**:
  - *Hydrodynamic interactions (HI).* As mentioned in [AS13], HI between particles moving inside a fluid has a great impact on their net movement. This could be an interesting point of research - to see if the components included in the process of the central dogma are affected, potentially affecting the dynamic behaviour of the synthetic genetic devices.
  - *Variable reaction radii* would affect the displacement calculated at each time step, given by the thermodynamic model formalised in Equation 3.7 in Chapter 3. This could e.g. result in stationary mRNA strands whilst the regulated proteins move more freely.
  - Motion of complex structures could also be included by describing the mRNA strand as chains of linked spheres instead of simple individual spheres.

Appendix A

negdevice.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2/version3" level="2" version="3">
  <model>
    <listOfCompartments>
      <compartment id="compartment1" spatialDimensions="3.0" size="100"/>
    </listOfCompartments>
    <listOfSpecies>
      <species id="Plac" compartment="compartment1" initialAmount="1" hasOnlySubstanceUnits="
          true"/>
      <species id="mRNA" compartment="compartment1" initialAmount="0" hasOnlySubstanceUnits="
          true"/>
      <species id="Lacl" compartment="compartment1" initialAmount="0" hasOnlySubstanceUnits="
          true"/>
      <species id="Plac_Lacl" compartment="compartment1" initialAmount="0"
          hasOnlySubstanceUnits="true"/>
    </listOfSpecies>
    <listOfReactions>
      <reaction id="transcription" reversible="false">
        <listOfProducts>
          <speciesReference species="mRNA"/>
        </listOfProducts>
        <listOfModifiers>
          <modifierSpeciesReference species="Plac"/>
        </listOfModifiers>
        <kineticLaw>
          <math xmlns="http://www.w3.org/1998/Math/MathML">
            <apply>
              <times/>
```

```
        <cn> 0.5 </cn>
        <ci> Plac </ci>
      </apply>
    </math>
  </kineticLaw>
</reaction>
<reaction id="regulation" reversible="false">
  <listOfProducts>
    <speciesReference species="Plac_Lacl"/>
  </listOfProducts>
  <listOfReactants>
    <speciesReference species="Plac"/>
    <speciesReference species="Lacl"/>
  </listOfReactants>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <cn> 1.0 </cn>
        <ci> Plac </ci>
        <ci> Lacl </ci>
      </apply>
    </math>
  </kineticLaw>
</reaction>
<reaction id="unbinds" reversible="false">
  <listOfReactants>
    <speciesReference species="Plac_Lacl"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species="Plac"/>
    <speciesReference species="Lacl"/>
  </listOfProducts>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <cn> 9.0 </cn>
        <ci> Plac_Lacl </ci>
      </apply>
    </math>
  </kineticLaw>
</reaction>
<reaction id="repressed_transcription" reversible="false">
  <listOfProducts>
    <speciesReference species="mRNA"/>
  </listOfProducts>
  <listOfModifiers>
    <modifierSpeciesReference species="Plac_Lacl"/>
  </listOfModifiers>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <cn> 0.0005 </cn>
```

```
            <ci> Plac_LacI </ci>
          </apply>
        </math>
      </kineticLaw>
    </reaction>
    <reaction id="translation" reversible="false">
      <listOfProducts>
        <speciesReference species="LacI"/>
      </listOfProducts>
      <listOfModifiers>
        <modifierSpeciesReference species="mRNA"/>
      </listOfModifiers>
      <kineticLaw>
        <math xmlns="http://www.w3.org/1998/Math/MathML">
          <apply>
            <times/>
            <cn> 0.167 </cn>
            <ci> mRNA </ci>
          </apply>
        </math>
      </kineticLaw>
    </reaction>
    <reaction id="decay_mRNA" reversible="false">
      <listOfReactants>
        <speciesReference species="mRNA"/>
      </listOfReactants>
      <kineticLaw>
        <math xmlns="http://www.w3.org/1998/Math/MathML">
          <apply>
            <times/>
            <cn> 0.0058 </cn>
            <ci> mRNA </ci>
          </apply>
        </math>
      </kineticLaw>
    </reaction>
    <reaction id="decay_LacI" reversible="false">
      <listOfReactants>
        <speciesReference species="LacI"/>
      </listOfReactants>
      <kineticLaw>
        <math xmlns="http://www.w3.org/1998/Math/MathML">
          <apply>
            <times/>
            <cn> 0.0012 </cn>
            <ci> LacI </ci>
          </apply>
        </math>
      </kineticLaw>
    </reaction>
  </listOfReactions>
 </model>
</sbml>
```

Appendix B

andgatedevice.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2/version3" level="2" version="3">
  <model>
    <listOfCompartments>
      <compartment id="compartment1" spatialDimensions="3.0" size="100"/>
    </listOfCompartments>
    <listOfSpecies>
      <species id="Pro" compartment="compartment1" initialAmount="0" hasOnlySubstanceUnits="
          true"/>
      <species id="lacI" compartment="compartment1" initialAmount="100" hasOnlySubstanceUnits=
          "true"/>
      <species id="IPTG" compartment="compartment1" initialAmount="100" hasOnlySubstanceUnits=
          "true"/>
      <species id="Ara" compartment="compartment1" initialAmount="0" hasOnlySubstanceUnits="
          true"/>
      <species id="mRNA_Ara" compartment="compartment1" initialAmount="0"
          hasOnlySubstanceUnits="true"/>
    </listOfSpecies>
    <listOfReactions>
      <reaction id="IPTG_lacI_p" reversible="false">
        <listOfProducts>
          <speciesReference species="Pro"/>
        </listOfProducts>
        <listOfModifiers>
          <modifierSpeciesReference species="IPTG"/>
          <modifierSpeciesReference species="lacI"/>
        </listOfModifiers>
        <kineticLaw>
```

```
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <cn> 0.0001 </cn>
          <ci> IPTG </ci>
          <ci> lacI </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
  <reaction id="tc_Ara" reversible="false">
    <listOfReactants>
      <speciesReference species="Pro"/>
    </listOfReactants>
    <listOfProducts>
      <speciesReference species="mRNA_Ara"/>
    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <cn> 0.012 </cn>
          <ci> Pro </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
  <reaction id="tl_Ara" reversible="false">
    <listOfProducts>
      <speciesReference species="Ara"/>
    </listOfProducts>
    <listOfModifiers>
      <modifierSpeciesReference species="mRNA_Ara"/>
    </listOfModifiers>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <cn> 0.009 </cn>
          <ci> mRNA_Ara </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
  <reaction id="decay_mRNA_Ara" reversible="false">
    <listOfReactants>
      <speciesReference species="mRNA_Ara"/>
    </listOfReactants>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <cn> 0.01 </cn>
          <ci> mRNA_Ara </ci>
        </apply>
```

```
        </math>
      </kineticLaw>
    </reaction>
    <reaction id="decay_Ara" reversible="false">
      <listOfReactants>
        <speciesReference species="Ara"/>
      </listOfReactants>
      <kineticLaw>
        <math xmlns="http://www.w3.org/1998/Math/MathML">
          <apply>
            <times/>
            <cn> 0.01 </cn>
            <ci> Ara </ci>
          </apply>
        </math>
      </kineticLaw>
    </reaction>
  </listOfReactions>
 </model>
</sbml>
```

Appendix C

ChemicalSystemModel.fs

```
module ChemicalSystemModel

open System

type Species = {
    name: string option;
    compartId: string;
    initAmount: float option;
    initConc: float option;
    substanceUnits: string option; //IdRef //differs from doc => not option
    hasOnlySubstanceUnits: bool;
    boundaryCond: bool option; //differs from doc => not option
    constant: bool option; //differs from doc => not option
    converFactor: string option}

type Compartment = {
    name: string option;
    spatialDim: float option;
    size: int option //size="1e-14"
    units: string option //IdRef
    constant: bool option} //differs from doc => not option

type LocalParameter = {
    name: string option;
    value: float option;
    units: string option}

type Variable = | Numerical of float
```

```
              | VarName of string

type Law = {
    func: string;
    variables: List<Variable>;
    }

type SpecRef = {stoichiometry: float option; constant: bool option} //constant option differs
      from doc

type ReactionAgent = | Reactants of Map<string,SpecRef>
                     | Products of Map<string,SpecRef>
                     | Modifiers of List<string>

type ReactionDesc = {
    name: string option;
    reversible: bool;
    fast: bool option; //differs from doc => not option
    compartment: string option; //IdRef
    kineticLaw: Law}

type Reaction = ReactionDesc * List<ReactionAgent>

type Part = | AllCompartments of Map<string,Compartment>
            | AllSpecies of Map<string,Species>
            | AllReactions of Map<string,Reaction>
            | Undefined

type Model = List<Part>
```

Appendix D

Parser.fsy

```
%{
open ChemicalSystemModel
open System
%}

%token QMARK XML VER ENCD SBML XMLNS LEVEL MODEL EQ
%token COMPARTS SPECS RECS ID NAME SPARDIM SIZE UNITS CONST COMPART SPECIES REACTION INITAMT
%token HASSUB BOUNDCOND INITCONC SUBSUNIT CONSTANT REVERS FAST CONVFACT
%token REACTANTS PRODUCTS MODIFS SPECREF MODREF STOICH VALUE
%token KLAW MATH APPLY TIMES CN CI URL
%token <string> STRING
%token <int> INT
%token <float> FLOAT
%token QUOTE STARTTAG ENDTAG END EQUALS
%token EOF

%start Main
%type <Model> Main

%%

Main:
    XmlHeader SBMLHeader ModelStart Model SBMLEnd EOF { $4 }

ModelStart: STARTTAG MODEL ENDTAG {}

ModelEnd: STARTTAG END MODEL ENDTAG {}
SBMLEnd: STARTTAG END SBML ENDTAG {}
```

```
XmlHeader:
    STARTTAG QMARK XML VER EQ FLOAT ENCD EQ STRING QMARK ENDTAG {}

SBMLHeader:
    STARTTAG SBML XMLNS EQ URL LEVEL EQ INT VER EQ INT ENDTAG {}

Model:
     Part ModelEnd     { [$1] }
    | Part Model  { $1 :: $2 }

Part:        { }
    | STARTTAG COMPARTS ENDTAG CompMap STARTTAG END COMPARTS ENDTAG  { AllCompartments(Map.
       ofList $4) }
    | STARTTAG SPECS ENDTAG SpecMap STARTTAG END SPECS ENDTAG   { AllSpecies(Map.ofList $4) }
    | STARTTAG RECS ENDTAG RecMap STARTTAG END RECS ENDTAG    { AllReactions(Map.ofList $4) }

UndefPart: STARTTAG StringList ENDTAG { }

UndefComponent: STARTTAG StringList END ENDTAG { }

StringList:
    |     { }
    | StringList STRING { }

UndfComponentList:
    |           { }
    | UndfComponentList UndefComponent  { }

CompMap:
                             { [ ]}
    | CompMap Compartment  { $2 :: $1 }

Compartment: STARTTAG COMPART ID EQ STRING CompartParas END ENDTAG { $5, $6 }

CompartParas:
     Name SpaDim Size Units Constant { {name = $1; spatialDim = $2; size = $3; units = $4;
          constant = $5} }

SpecMap:
                         { [ ]}
    | SpecMap Species  { $2 :: $1 }

Species: STARTTAG SPECIES ID EQ STRING SpeciesParas END ENDTAG { $5, $6 }

SpeciesParas:
     Name COMPART EQ STRING InitAmt InitConc
     SubsUnits HASSUB EQ STRING BoundCond Constant ConvFact { {name = $1; compartId = $4;
          initAmount = $5;
                                                          initConc = $6; substanceUnits = $7;
                                                          hasOnlySubstanceUnits = Boolean.
                                                               Parse($10);
                                                          boundaryCond = $11; constant = $12;
                                                          converFactor = $13} }
```

```
RecMap:
    |      { [ ] }
    | RecMap Reaction   { $2 :: $1 }

Reaction: STARTTAG REACTION ID EQ STRING Name REVERS EQ STRING Fast CompartRef ENDTAG
         AgentList
         STARTTAG KLAW ENDTAG KinecticLaw STARTTAG END KLAW ENDTAG
         STARTTAG END REACTION ENDTAG { $5, ({name = $6; reversible = Boolean.Parse($9); fast
               = $10;
                                                    compartment = $11; kineticLaw = $17
                                                    }, $13) }

AgentList:
    |       { [] }
    | AgentList ReactionAgent { $2 :: $1 }

ReactionAgent:
    | STARTTAG REACTANTS ENDTAG RefMap STARTTAG END REACTANTS ENDTAG { Reactants($4 |> Map.
        ofList) }
    | STARTTAG PRODUCTS ENDTAG RefMap STARTTAG END PRODUCTS ENDTAG  { Products($4 |> Map.
        ofList) }
    | STARTTAG MODIFS ENDTAG RefList STARTTAG END MODIFS ENDTAG   { Modifiers($4) }

RefMap:
    | SpecRef    { [$1] }
    | RefMap SpecRef   { $2 :: $1 }

SpecRef: STARTTAG SPECREF SPECIES EQ STRING SpecRefParas END ENDTAG { $5, $6 }

RefList:
    | ModRef    { [$1] }
    | RefList ModRef  { $2 :: $1 }

ModRef: STARTTAG MODREF SPECIES EQ STRING END ENDTAG     { $5 }

SpecRefParas:
     Stoich Constant { {stoichiometry = $1; constant = $2} }

KinecticLaw:
    STARTTAG MATH XMLNS EQ URL ENDTAG
    STARTTAG APPLY ENDTAG
    STARTTAG TIMES END ENDTAG
    Variables
    STARTTAG END APPLY ENDTAG
    STARTTAG END MATH ENDTAG { {func = "apply"; variables = $14} }

Numericals:
    | Num     { [ $1 ] }
    | Numericals Num { $2 :: $1 }

Num: STARTTAG CN ENDTAG FLOAT STARTTAG END CN ENDTAG { $4 }

Variables:
    |     { [ ] }
    | Variables Var  { $2 :: $1 }
```

```
Var:
    | STARTTAG CI ENDTAG STRING STARTTAG END CI ENDTAG { VarName($4) }
    | STARTTAG CN ENDTAG FLOAT STARTTAG END CN ENDTAG { Numerical($4) }

InitAmt:
                        { None }
    | INITAMT EQ INT  { Some(float($3)) }
    | INITAMT EQ FLOAT  { Some($3) }

InitConc:
                        { None }
    | INITCONC EQ FLOAT { Some($3) }

SubsUnits:
                        { None }
    | SUBSUNIT EQ STRING { Some($3) }

BoundCond:
                        { None }
    | BOUNDCOND EQ STRING { Some(Boolean.Parse($3)) }

ConvFact:
                        { None }
    | CONVFACT EQ STRING { Some($3) }

Name:
                        { None }
    | NAME EQ STRING  { Some($3) }

SpaDim:
                        { None }
    | SPARDIM EQ FLOAT  { Some($3) }

Size:
                        { None }
    | SIZE EQ INT   { Some($3) }

Constant:
                        { None }
    | CONSTANT EQ STRING { Some(Boolean.Parse($3)) }

Fast:
                        { None }
    | FAST EQ STRING  { Some(Boolean.Parse($3)) }

CompartRef:
                        { None }
    | COMPART EQ STRING { Some($3) }

Units:
                        { None }
    | UNITS EQ STRING  { Some($3) }

Stoich:
```

```
                    { None }
  | STOICH EQ INT  { Some(float($3)) }
  | STOICH EQ FLOAT  { Some($3) }
```

Appendix E

Lexer.fsl

```
{
module Lexer
open System
open System.Text
open Parser
open Microsoft.FSharp.Text.Lexing

let keyword s =
    match s with
    | "xml"       -> XML
    | "version"     -> VER
    | "encoding"    -> ENCD
    | "sbml"     -> SBML
    | "model"     -> MODEL
    | "xmlns"     -> XMLNS
    | "level"     -> LEVEL
    | "listOfCompartments"  -> COMPARTS
    | "listOfSpecies"   -> SPECS
    | "listOfReactions"   -> RECS
    | "id"       -> ID
    | "name"     -> NAME
    | "spatialDimensions"  -> SPARDIM
    | "size"     -> SIZE
    | "units"     -> UNITS
    | "compartment"    -> COMPART
    | "species"     -> SPECIES
    | "reaction"    -> REACTION
    | "initialAmount"   -> INITAMT
```

```
    | "initialConcentration" -> INITCONC
    | "substanceUnits"   -> SUBSUNIT
    | "hasOnlySubstanceUnits" -> HASSUB
    | "boundaryCondition"  -> BOUNDCOND
    | "constant"    -> CONSTANT
    | "converFactor"   -> CONVFACT
    | "reversible"    -> REVERS
    | "fast"     -> FAST
    | "listOfReactants"   -> REACTANTS
    | "listOfProducts"   -> PRODUCTS
    | "listOfModifiers"   -> MODIFS
    | "kineticLaw"    -> KLAW
    | "value"     -> VALUE
    | "speciesReference"  -> SPECREF
    | "modifierSpeciesReference"-> MODREF
    | "stoichiometry"   -> STOICH
    | "math"     -> MATH
    | "apply"     -> APPLY
    | "times"     -> TIMES
    | "cn"      -> CN
    | "ci"      -> CI
    | _       -> STRING s

}

let digit  = ['0'-'9']
let int   = '-'?digit+
let float  = '-'?digit+ '.' digit+
let whitespace = [' ' '\t' ]
let newline  = ('\n' | '\r' '\n')
let letter  = ['A'-'Z' 'a'-'z']
let identifier = letter(letter|digit|float|['_']|['-'])*
let escapes  = ''' | '\"' | '.' | '!' | '-' | '_'
let comment  = "<!--" ([^'-'] [^'-'] [^'>'])*
let quote = '\"'
let url   = quote "http" ([^'\"'])* quote

rule tokenize = parse
| "<"     { STARTTAG }
| ">"     { ENDTAG }
| "?"     { QMARK }
| '='     { EQ }
| whitespace   { tokenize lexbuf }
| newline    { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf }
| comment    { tokenize lexbuf }
| url     { URL}
| int     { INT<| Int32.Parse(Encoding.UTF8.GetString(lexbuf.Lexeme)) }
| float     { FLOAT <| float (Encoding.UTF8.GetString(lexbuf.Lexeme)) }
| "/"     { END }
| escapes    { tokenize lexbuf }
| identifier   { let s = Encoding.UTF8.GetString(lexbuf.Lexeme);
                         keyword(s) }
| eof     { EOF }
```

Appendix F

ParserUtil.fs

```
module ParserUtil

open System.IO
open System.Text
open Microsoft.FSharp.Text.Lexing

open ChemicalSystemModel
open Lexer
open Parser

let parseString (text:string) =
   let lexbuf = LexBuffer<_>.FromBytes(Encoding.UTF8.GetBytes(text))
   try
       Parser.Main Lexer.tokenize lexbuf
   with e ->
        let pos = lexbuf.EndPos
        printfn "Error near line %d, character %d\n" pos.Line pos.Column
        failwith "parser termination"

let parseFromFile filename =
   if File.Exists(filename)
   then parseString(File.ReadAllText(filename))
   else invalidArg "ParserUtil" "File not found"
```

Appendix G

SPNbase.fs

```
module SPNbase

open Space
open Coordinate

type Tokens<'a> = 'a
type RateFunction = float
type Transition = string * RateFunction
type Place = string
type Arc = | TransPlace of Transition * Place
           | PlaceTrans of Place * Transition
           | Modifier of Place * Transition //extension - is always enabled and doesn't
                consume tokens upon fire

type SPN<'TokenCollection,'Token> = {
    marking : Map<Place,'TokenCollection>;
    transitions : Map<Transition,Arc List>;
    genTokens : int->'TokenCollection;
    optional : Option<Space>;
    tokensInPlace : Map<Place,'TokenCollection>->Place->int
    removeToken : 'Token->'TokenCollection->'TokenCollection
    fireRule : SPN<'TokenCollection,'Token>->Place list->Transition->Option<Space>
                   ->'Token option*Map<Place,'TokenCollection>
    addToken : Map<Place,'TokenCollection>->Place list->'Token->Map<Place,'TokenCollection>
    nextState : SPN<'TokenCollection,'Token>->float->SPN<'TokenCollection,'Token>
    }

let placesOut t transitions =
```

```
    Map.find t transitions |> List.fold (fun ls arc -> match arc with
                                                       | TransPlace(t1,p) when t1 = t -> p::ls
                                                       | _ -> ls) []

let placesIn t transitions =
    Map.find t transitions |> List.fold (fun ls arc -> match arc with
                                                       | PlaceTrans(p,t1) when t1 = t -> p::ls
                                                       | Modifier(p,t1) when t1 = t -> p::ls
                                                       | _ -> ls) []

let rec isModifier place t spn =
    Map.find t spn.transitions |> List.fold (fun isMod arc ->match arc with
                                                        | Modifier(p,t1) when t1 = t && p =
                                                            place-> true
                                                        | _ -> isMod) false

let canFire spn t =
    placesIn t spn.transitions |> List.fold (fun canfire p -> spn.tokensInPlace spn.marking p
        > 0 && canfire) true

let removeTokenFromPlace place trans token spn =
    if not (isModifier place trans spn) then spn.marking |> Map.map (fun p (tks:'a) -> if p =
        place then spn.removeToken token tks else tks)
                                    else spn.marking

let fire spn t =
    if canFire spn t then
       let pIn = placesIn t spn.transitions
       let (token,newMarking) = spn.fireRule spn pIn t spn.optional
       let tempspn = {spn with marking = newMarking}
       match token with
       | Some tok -> let firedMarking = spn.addToken tempspn.marking (placesOut t tempspn.
            transitions) tok
                     { tempspn with marking = firedMarking }
       | None -> spn
    else spn

//auxilliary functions for simulation
let genNextState spn = spn.nextState spn

let state spn = spn.marking |> Map.map (fun p t -> spn.tokensInPlace spn.marking p)

let getTransition trans spn =
    spn.transitions |> Map.fold (fun t (name,haz) arcs -> if name = trans then (name,haz) else
         t) ("",0.0)

let isOfInterest trans (placesOfInterest:string list) spn =
    placesOfInterest |> List.fold (fun exists p -> (placesOut trans spn.transitions) |> List.
        exists (fun p1 -> p = p1) || exists) false

let h spn t =
    let places = placesIn t spn.transitions
    spn.marking |> Map.fold (
       fun prod p1 t -> if places |> List.exists(fun p2 -> p1 = p2) then prod * float(spn.
            tokensInPlace spn.marking p1)
```

```
                    else prod) 1.0

let a spn = spn.transitions |> Map.fold (fun ls (t1,hazard) arcs -> (t1,hazard * h spn (t1,
    hazard))::ls) []

//for debugging
let tokensInplace place spn = List.length (Map.find place spn.marking)
```

Appendix H

SPNint.fs

```
module SPNint

open SPNbase

let makeSPNint : SPN<int,int> =
    {
    marking = Map.ofList [];
    transitions = Map.ofList [];
    genTokens = (fun i -> i);
    optional = None;
    tokensInPlace = (fun marking place -> marking |> Map.find place);
    removeToken = (fun _ i -> i-1);
    fireRule = (fun spn places t opt ->
                      (Some(1),spn.marking |> Map.map (fun p tks ->
                          if List.exists (fun p1 -> p1 = p) places then
                              if isModifier p t spn then tks else tks-1
                          else tks)));
    addToken = (fun marking places token ->
                       marking |> Map.map (fun p tks -> if List.exists (fun p1 -> p1 = p)
                            places then tks+1 else tks))
    nextState = (fun spn deltaT -> spn)
    }
```

Appendix I

SPNlist.fs

```
module SPNlist

open Coordinate
open Space
open SPNbase

let fireTransition spn places t space =
    let placesWithTokens = spn.marking |> Map.filter (fun p tks -> List.exists (fun p1 -> p1 =
          p) places) |> Map.toList
    let p = fst (List.head placesWithTokens)
    let (t1,_) = pickRandomFromList (snd (List.head placesWithTokens))
    match List.length placesWithTokens with
    | 1 -> (Some (t1),removeTokenFromPlace p t t1 spn)
    | 2 -> let n = findneighbours (snd (List.nth placesWithTokens 0)) (snd (List.nth
         placesWithTokens 1)) (Option.get space)
           let p2 = fst (List.nth placesWithTokens 1)
           match n with
           | [] -> (None, spn.marking) //no neighbouring particles were found and we should
                not consume tokens
           | [t1;t2] -> let newspn = {spn with marking = removeTokenFromPlace p t t1 spn}
                        (Some (meanPosition n), removeTokenFromPlace p2 t t2 newspn)
           | _ -> failwith "error: not ment to find more than two neighbours"
    | _ -> failwith "error: got too many places pointing to transition - abstracted to atmost
         two"

let putTokens marking places token = marking |> Map.map (fun p tks ->if List.exists (fun p1
     -> p1 = p) places
                                                                then token::tks
```

```
                                                        else tks)

let rec removeTokenFromList (token:Coordinate) (tokens:Coordinate list) : Coordinate list =
    let rec remove tks =
        match tks with
        | t::ts when t = token -> ts
        | t::ts -> t::remove ts
        | _ -> []
    remove tokens

let makeSPNlist space : SPN<Coordinate list,Coordinate> =
    {
    marking = Map.ofList [];
    transitions = Map.ofList [];
    genTokens = (fun i -> List.init i (fun _ -> genCoordinate space));
    optional = Some(space);
    tokensInPlace = (fun marking place -> List.length (Map.find place marking));
    removeToken = removeTokenFromList;
    fireRule = fireTransition;
    addToken = putTokens
    nextState = (fun spn deltaT ->
                    {spn with marking = spn.marking |> Map.map (fun p tks -> applyMoves (
                        List.init tks.Length (fun _ -> genMove (Option.get spn.optional)
                        deltaT)) tks [] (Option.get spn.optional)) })
    }
```

Appendix J

SPNarray.fs

```
module SPNarray

open Coordinate
open Space
open SPNbase

let removeTokenFromArray token (tokens:Coordinate array) = //this function is not robust,
     since we imply that t exists in tokens
    let t = let randomindex = rnd.Next(Array.length tokens - 1)
            tokens.[randomindex]
    let rec initArray (array:Coordinate array) newi oldi skipped max =
        match newi < max with
        | true -> if tokens.[oldi] = t && skipped then array.[newi] <- tokens.[oldi];
             initArray array (newi+1) (oldi+1) skipped max
                   elif tokens.[oldi] = t && not skipped then array.[newi] <- tokens.[(oldi+1)
                        ]; initArray array (newi+1) (oldi+2) true max
                   else array.[newi] <- tokens.[oldi]; initArray array (newi+1) (oldi+1)
                        skipped max
        | false -> array
    let dummyToken = {x = System.Double.MaxValue; y =System.Double.MaxValue; z = System.Double
         .MaxValue}
    let newLength = Array.length tokens - 1
    let newArray = Array.create newLength dummyToken
    initArray newArray 0 0 false (Array.length newArray)

//removes a token from each place and returns the new marking and mean position of the tokens
      that were removed
let fireTransition spn places t space =
```

```
    let placesWithTokens = spn.marking |> Map.filter (fun p tks -> List.exists (fun p1 -> p1 =
          p) places) |> Map.toList
    let p = fst (List.head placesWithTokens)
    let tokens:Coordinate array = (snd (List.head placesWithTokens))
    let (t1,_) = pickRandomFromArray tokens//is returned in case the transition only has one
         ingoing place
    match List.length placesWithTokens with
    | 1 -> (Some (t1),removeTokenFromPlace p t t1 spn)
    | 2 -> let (tokens1,tokens2) = ((snd (List.nth placesWithTokens 0)),(snd (List.nth
         placesWithTokens 1)))
           let (longest,shortest) =
                if (Array.length tokens1) >= (Array.length tokens2) then (tokens1,tokens2) else
                     (tokens2,tokens1)
           let n = findneighboursarray shortest longest (Option.get space)
           let p2 = fst (List.nth placesWithTokens 1)
           match n with
           | [] -> (None, spn.marking) //no neighbouring particles were found and we should
                not consume tokens
           | [t1;t2] -> let newspn = {spn with marking = removeTokenFromPlace p t t1 spn}
                        (Some (meanPosition n), removeTokenFromPlace p2 t t2 newspn)
           | _ -> failwith "error: not ment to find more than two neighbours"
    | _ -> failwith "error: got too many places pointing to transition - abstracted to atmost
         two"

let putTokens marking places token = marking |> Map.map (fun p tks ->if List.exists (fun p1
     -> p1 = p) places
                                                                then Array.append [|token|] tks
                                                                else tks)

let tokensInPlace1 marking place = Array.length (Map.find place marking)

let makeSPNarray space : SPN<Coordinate array,Coordinate> =
    {
    marking = [] |> Map.ofList;
    transitions = [] |> Map.ofList;
    genTokens = (fun i -> Array.init i (fun _ -> genCoordinate space));
    optional = Some(space);
    tokensInPlace = (fun marking place -> Array.length (Map.find place marking));
    removeToken = removeTokenFromArray;
    fireRule = fireTransition;
    addToken = putTokens
    nextState = (fun spn deltaT ->
                     {spn with marking = spn.marking |>
                         Map.map (fun p tks -> let moves = Array.init tks.Length (fun _ ->
                              genMove (Option.get spn.optional) deltaT)
                                               let newtks = applyMovesArray moves tks (Option.
                                                    get spn.optional)
                                               newtks) })
    }
```

Appendix K

BrownianMotion.fs

```
module BrownianMotion

open MathNet.Numerics.Distributions
open Coordinate

type Motion =
    | Constant of float
    | VelocityAtTemp of (float->float)
    | VelocityAtTempRad of (float->float->float)

type Distribution =
    | UniformDist of ContinuousUniform
    | NormalDist of Normal

let pi = System.Math.PI
let kB = 1.3806488e-23 //Boltzmann's constant
let eta = 8.0 //cytoplasm has 8 times the viscosity
let alpha r = 6.0*pi*eta*r

let direction n = if n % 2 = 0 then 1.0 else -1.0

let genNoiseVector (distribution:Distribution) =
    let (xdir:float,ydir:float,zdir:float) =
        match distribution with
        | NormalDist(normal) -> (normal.Sample(),normal.Sample(),normal.Sample())
        | UniformDist(rnd) -> (rnd.Sample() * (direction (int (rnd.Sample()*10.0))),
                               rnd.Sample() * (direction (int (rnd.Sample()*10.0))),
                               rnd.Sample() * (direction (int (rnd.Sample()*10.0))))
```

```
    {x = xdir; y = ydir; z = zdir}

let applyMotion motion temp radius =
    match motion with
    | Constant(f) -> f
    | VelocityAtTemp(f) -> f temp / sqrt 3.0 //for vector |v| = sqrt 3 * distance
    | VelocityAtTempRad(f) -> f temp radius / sqrt 3.0
```

Appendix L

Space.fs

```
module Space
open System
open BrownianMotion
open Coordinate
open MathNet.Numerics.Distributions

let rnd = System.Random()
let (mean,stddev) = (0.0,0.25)
let normalDist = new Normal(mean, stddev)

//s is in nanometres, temperature is in Kelvin
//elasticity is the energyabsorbtion of membrane interaction
type Space = {
    size : float;
    radius : float;
    temperature : float;
    motion : Motion;
    distribution : Distribution;
    elasticity : float}

let genCoordinate dim =
    {x = dim.size; y = dim.size; z = dim.size} / 2.0

let rec boundary x space =
    if x < space.size && x > 0.0 then x
    elif x < 0.0 then boundary (-x*space.elasticity) space
    else
        let left = x - space.size * space.elasticity
```

```
        boundary (space.size-left) space

let genMove space deltaT =
    let noise = genNoiseVector space.distribution
    let speed = applyMotion space.motion space.temperature space.radius
    let distance = speed * deltaT
    distance * noise

let applyMove space coordinate move =
    {x = boundary (coordinate.x + move.x) space;
    y = boundary (coordinate.y + move.y) space;
    z = boundary (coordinate.z + move.z) space}

let rec applyMoves moves coords newCoords space =
    match moves with
    | [] -> newCoords
    | m::ms -> match coords with
               | [] -> newCoords
               | c::cs -> let newCoord = (applyMove space c m)
                          applyMoves ms cs (newCoord::newCoords) space

let applyMovesArray (moves:Coordinate array) coords space =
    coords |> Array.mapi (fun i c -> applyMove space c moves.[i])

let inRange c1 c2 space =
    let xd = c1.x-c2.x
    let yd = c1.y-c2.y
    let zd = c1.z-c2.z
    sqrt(xd*xd + yd*yd + zd*zd) < space.radius

let meanPosition coords =
    let sum = coords |> List.fold (fun s c -> s + c) {x = 0.0; y = 0.0; z = 0.0}
    let nrOfCoords = float (List.length coords)
    sum / nrOfCoords

let pickRandomFromList particles =
    let choose = rnd.Next(List.length particles - 1)
    let p = particles.[choose]
    let particles = particles |> List.filter (fun p1 -> not(p1 = p))
    (p,particles)

let rec findneighbours tokens1 tokens2 space =
    match tokens1 with
    | [] -> []
    | _ -> let (ta,tas) = pickRandomFromList tokens1
               let neighbour = tokens2 |> List.tryFind(fun t -> inRange ta t space)
               match neighbour with
               | None -> findneighbours tas tokens2 space
               | Some(tb) -> [ta;tb]

let pickRandomFromArray particles =
    let choose = rnd.Next(Array.length particles - 1)
    let p = particles.[choose]
    let particles = particles |> Array.filter (fun p1 -> not(p1 = p))
    (p,particles)
```

```
let rec findneighboursarray tokens1 tokens2 space = //assumes tokens2 is the largest
    match tokens1 with
    | [||] -> []
    | _ -> let (ta,tas) = pickRandomFromArray tokens1
                let neighbours = tokens2 |> Array.Parallel.choose (fun t -> if inRange ta t
                     space then Some(t) else None)
                match neighbours with
                | [||] -> findneighboursarray tas tokens2 space
                | _ -> [ta;neighbours.[rnd.Next(Array.length neighbours - 1)]]
```

Appendix M

SPNbaseCompiler.fs

```
module SPNbaseCompiler

open SPNbase
open ChemicalSystemModel
open Space

let getRate (law:Law) =
    List.fold (fun r v -> match v with
                        | Numerical(rate) -> rate
                        | VarName(_) -> r) 0.0 law.variables

let transRecAgent recAgent transition =
    match recAgent with
    | Reactants(rects) -> Map.fold (fun r rName _ -> PlaceTrans(rName,transition)::r) [] rects
    | Products(prods) -> Map.fold (fun r pName _ -> TransPlace(transition,pName)::r) [] prods
    | Modifiers(mods) -> List.fold (fun r mName -> Modifier(mName,transition)::r) [] mods

let transReactions reactions spn =
    Map.fold (fun s name (recDesc,recAgents) ->
        {spn with transitions = s.transitions.Add ((name,getRate recDesc.kineticLaw),
                                                   List.fold(fun a recAgent -> (transRecAgent
                                                        recAgent (name,getRate recDesc.kineticLaw)
                                                        )@a) [] recAgents);
         }) spn reactions

let transSpecies species spn =
    Map.fold (fun s name (species:Species) ->
```

```
        {spn with marking = s.marking.Add (name,match species.initAmount with Some (i) -> spn.
            genTokens (int i) | None -> spn.genTokens 0)
        }) spn species

let transCompart compartments spn =
    Map.fold (fun s name (compartment:Compartment) ->
        {spn with optional = match compartment.spatialDim with
                             | Some(3.0) -> match spn.optional with
                                            | Some (space) -> Some({space with size = float (
                                                Option.get compartment.size)}:Space)
                                            | None -> None
                             | _ -> printf "Please define the spatial dimensions.";None
        }) spn compartments

let transPart (p:Part) spn =
    match p with
    | AllSpecies(species) -> transSpecies species spn
    | AllReactions(reactions) -> transReactions reactions spn
    | AllCompartments(comparments) -> transCompart comparments spn
    | _ -> spn

let compileModel (spn:SPN<_,_>) (model:Model) =
    List.fold (fun s part -> transPart part s) spn model
```

Appendix N

Simulator.fs

```
module Simulator

open Microsoft.FSharp.Collections
open MathNet.Numerics.Distributions
open Statistics

type Data = (float*float) list list
type MinMaxData = ((float*float) list*(float*float) list) list
type Snapshots<'datastructure> = 'datastructure list
type SimulationResult<'datastructure> = ((Data * MinMaxData) * Snapshots<'datastructure>) *
     float list list

let rnd = new ContinuousUniform()

let formatData data s =
    data |> Array.fold (fun ls (t,marking) ->
        (t,marking |> Map.fold(fun r p tokens ->
            if p = s then float tokens else r) 0.0 )::ls) []

let formatSimulation names res formatter =
    names |> List.fold (fun ls n -> (formatter res n)::ls) []

let createTasks simulations maxTasks =
    let rest = (max simulations maxTasks) % (min simulations maxTasks)
    let simsPerTask = (max simulations maxTasks) / (min simulations maxTasks)
    if simulations > maxTasks then
        if rest = 0 then Array.init maxTasks (fun _ -> simsPerTask)
```

```
        else Array.init maxTasks (fun i -> if i = (maxTasks - 1) then simsPerTask+rest else
            simsPerTask)
    else Array.create simulations 1


let getLastMeasurements (data: (float*float) list list) : float list =
    List.init (List.length data) (fun i -> snd (List.head (List.nth data i)))

let simulate algo datastructure partsOfInterest snapInterval duration simulations maxTasks
    : SimulationResult<_> =
    let rec runSimulations spn count sims minmax partsOfInterest algo snapInterval snapshots
         lastmeasurements simulations =
        if count >= simulations then
            printf "simulation done\n";((((sims,minmax),snapshots),lastmeasurements),
                simulations)
        else
            let (snapshots,res) = algo spn partsOfInterest snapInterval duration rnd
            let formattedData = formatSimulation partsOfInterest res formatData
            let minmaxData = updateAllAccumilatedMinMax formattedData minmax []
            let averagedData = averageAllData formattedData sims [] count //used for plotting
                the average behaviour
            let lastmeasurements = (getLastMeasurements formattedData)::lastmeasurements //
                later used for statistical analysis
            runSimulations spn (count+1) averagedData minmaxData partsOfInterest algo
                snapInterval snapshots lastmeasurements simulations
    let tasks = createTasks simulations maxTasks
    printf "tasklist %A\n" tasks
    let results =
        Array.Parallel.map
            (runSimulations datastructure 0 [] [] partsOfInterest algo snapInterval [] [])
            tasks
        |> Array.toList
    let combinedResult =
        results |> List.reduce (fun ((((accData,accMinMax),snap),lastm1),sims1) ((((data,_),_)
             ,lastm2),sims2)
                                -> (((averageAllData data accData [] (sims1+sims2),
                                       updateAllAccumilatedMinMax data accMinMax []),
                                       snap),lastm1@lastm2),
                                       (sims1+sims2))
    fst combinedResult
```

Appendix O

Gillespie.fs & SpatialGillespie.fs

```
module Gillespie

open SPNbase
open MathNet.Numerics.Distributions

let gillespie spn partsOfInterest snapShotInterval maxT (rnd:MathNet.Numerics.Distributions.
     ContinuousUniform) =
    let result = Array.init maxT (fun _ -> (0.0,Map.ofList [("",0)]))
    let rec sim spn1 t i snapshots nextSnap lastSnapT =
        let aj = a spn1
        let a0 = aj |> List.fold (fun sum (t,a) -> sum + a) 0.0
        let r1 = rnd.Sample()
        let r2 = rnd.Sample()
        let tau = (1.0 / a0) * log (1.0 / r1)
        let a = aj |> List.filter (fun (_,ai) -> ai > r2 * a0 && ai <= a0)
        let (trans,_) = match a with
                        | [] -> aj |> List.minBy (fun (t,h) -> h)
                        | _ -> a |> List.minBy (fun (t,h) -> h)
        let transToFire = getTransition trans spn1
        let t = t + tau
        let newspn = fire spn1 transToFire
        let (nextSnap,snapshots,lastSnapT) =
            if t > nextSnap then
                (nextSnap + snapShotInterval,(newspn,lastSnapT)::snapshots,t-lastSnapT)
            else (nextSnap,snapshots,lastSnapT)
        if i < maxT-1 then
```

```
            if isOfInterest transToFire partsOfInterest newspn then
                result.[i] <- (t, state newspn)
                sim newspn t (i+1) snapshots nextSnap lastSnapT
            else sim newspn t i snapshots nextSnap lastSnapT
        else result.[i] <- (t, state newspn);snapshots
    (sim spn 0.0 0 [] 0.0 0.0,result)


module SpatialGillespie
open MathNet.Numerics.Distributions
open SPNbase

let spatial spn partsOfInterest snapShotInterval maxT (rnd:MathNet.Numerics.Distributions.
     ContinuousUniform) =
    let result = Array.init maxT (fun _ -> (0.0,Map.ofList [("",0)]))
    let rec sim spn1 t i snapshots nextSnap lastSnapT =
        let aj = a spn1
        let a0 = aj |> List.fold (fun sum (t,a) -> sum + a) 0.0
        let r1 = rnd.Sample()
        let r2 = rnd.Sample()
        let tau = (1.0 / a0) * log (1.0 / r1)
        let a = aj |> List.filter (fun (_,ai) -> ai > r2 * a0)
        let (trans,_) = match a with
                        | [] -> aj |> List.minBy (fun (t,h) -> h)
                        | _ -> a |> List.minBy (fun (t,h) -> h) //|> List.min
        let transToFire = getTransition trans spn1
        let firedspn = fire spn1 transToFire
        let movedspn = genNextState firedspn tau
        let t = t + tau
        let (nextSnap,snapshots,lastSnapT) =
            if t > nextSnap then
                (nextSnap + snapShotInterval,(movedspn,lastSnapT)::snapshots,t-lastSnapT)
            else (nextSnap,snapshots,lastSnapT)
        if i < maxT-1 then
            if isOfInterest transToFire partsOfInterest movedspn then
                result.[i] <- (t, state movedspn)
                sim movedspn t (i+1) snapshots nextSnap lastSnapT
            else sim movedspn t i snapshots nextSnap lastSnapT
        else result.[i] <- (t, state movedspn);snapshots
    (sim spn 0.0 0 [] 0.0 0.0,result)
```

Appendix P

Statistics.fs

```
module Statistics

open MathNet.Numerics.Statistics
open SPNbase
open Coordinate

let localmaxTime data = data |> List.fold (fun m (time,_) -> max m time) 0.0
let localmaxAmount data = data |> List.fold (fun m (_,amt) -> max m amt) 0.0
let globalMaxofData data =
    data |> List.fold (fun (maxT,maxA) d
                      -> (max maxT (localmaxTime d), max maxA (localmaxAmount d)))
                          (0.0,0.0)

let maxFloatVal = System.Double.MaxValue
let minDummy = {x = maxFloatVal; y = maxFloatVal; z = maxFloatVal}
let maxDummy = {x = 0.0; y = 0.0; z = 0.0}

let minmaxofCoords coords =
    let (minx,miny,minz,maxx,maxy,maxz) =
        coords |> List.fold (fun (cminx,cminy,cminz,cmaxx,cmaxy,cmaxz) c
                              -> (Coordinate.minX(cminx,c),
                                  Coordinate.minY(cminy,c),
                                  Coordinate.minZ(cminz,c),
                                  Coordinate.maxX(cmaxx,c),
                                  Coordinate.maxY(cmaxy,c),
                                  Coordinate.maxZ(cmaxz,c)))
                                  (minDummy,minDummy,minDummy,maxDummy,maxDummy,maxDummy)
    (minx.x,miny.y,minz.z,maxx.x,maxy.y,maxz.z)
```

```
let findMinMaxCoords snaps =
    let allCoords =
        snaps |> List.fold (fun ls (spn,_) -> (spn.marking |> Map.fold (fun pls p tks -> (
            Array.toList tks)@pls) [])@ls) []
    minmaxofCoords allCoords

let confidenceInterval (data: float list list) indexOfInterest : float * float =
    let measurements = data |> List.fold (fun ls entry -> (List.nth entry indexOfInterest)::ls
        ) []
    let sampleSize = List.length measurements
    let criticalVal = 1.96 // z0.025 - confidence of 95%
    let standardDeviation = Statistics.StandardDeviation measurements
    let sampleMean = Statistics.Mean measurements
    let standardError = standardDeviation / sqrt (float sampleSize)
    let marginOfError = criticalVal * standardError
    (sampleMean - marginOfError, sampleMean + marginOfError)

let rec averageData n data1 avg res =
       match data1,avg with
       | ((amt1,t1)::ds,(amt2,t2)::bs) -> let avgAmt = (amt1+n*amt2)/(n+1.0)
                                          let avgT = (t1+n*t2)/(n+1.0)
                                          averageData n ds bs ([(avgAmt,avgT)]@res)
       | _ -> res

let sortByTime (t1,_) (t2,_) = if t1 < t2 then 1 else -1

let rec averageAllData data avg res n =
    match data,avg with
    | (d::ds,a::als) -> averageAllData ds als (List.rev ((averageData (float n) d a [] |> List
        .sortWith sortByTime)::res)) n
    | (d::ds,[]) -> data
    | _ -> res

let getMinMax (meas:float*float) (mins:float*float) (maxs:float*float) =
    match meas,mins,maxs with
    | (amt1,t1),(minamt,mint),(maxamt,maxt) -> if amt1 < minamt then ((amt1,t1),maxs)
                                               elif amt1 > maxamt then (mins,(amt1,t1))
                                               else (mins,maxs)

let rec accumilated d acc =
    match d with
    | (amt,_)::ds -> accumilated ds (acc+amt)
    | [] -> acc

let getAccumilatedMinMax data mins maxs =
    let newacc = accumilated data 0.0
    let minacc = accumilated mins 0.0
    let maxacc = accumilated maxs 0.0
    if newacc < minacc then (data,maxs)
    elif newacc > maxacc then (mins,data)
    else (mins,maxs)

let rec updateMinMax data mins maxs resmins resmaxs : ((float*float)list*(float*float)list) =
    match data,mins,maxs with
```

```
    | (d::ds,mina::minss,maxa::maxss) -> let (newmin,newmax) = getMinMax d mina maxa
                                       updateMinMax ds minss maxss (newmin::resmins) (newmax::
                                           resmaxs)
    | _ -> (resmins |> List.sortWith sortByTime,resmaxs |> List.sortWith sortByTime)

let rec updateAllAccumilatedMinMax data minmax res =
    match data,minmax with
    | (d::ds,(mins,maxs)::ms) -> updateAllAccumilatedMinMax ds ms (List.rev ((
        getAccumilatedMinMax d mins maxs)::res))
    | (d::ds,[]) -> updateAllAccumilatedMinMax ds minmax (List.rev ((getAccumilatedMinMax d d
        d)::res))
    | _ -> res

let rec updateAllMinMax data minmax res =
    match data,minmax with
    | (d::ds,(mins,maxs)::ms) -> updateAllMinMax ds ms (List.rev ((updateMinMax d mins maxs []
         [])::res))
    | (d::ds,[]) -> updateAllMinMax ds minmax (List.rev ((updateMinMax d d d [] [])::res))
    | _ -> res
```

Appendix Q

DataWriter.fs

```
module DataWriter

open Coordinate
open Space
open SPNbase
open System.IO
open System.Runtime.Serialization.Formatters.Binary

let saveValue v path =
    use fsOut = new FileStream(path,FileMode.Create)
    let formatter = new BinaryFormatter()
    formatter.Serialize(fsOut,box v)
    fsOut.Close()

let restoreValue path =
    use fsIn = new FileStream(path,FileMode.Open)
    let formatter = new BinaryFormatter()
    let res = formatter.Deserialize(fsIn)
    fsIn.Close()
    unbox res

let paramsToString rs sims sp =
    "" + string rs + "x" + string sims + "x" + string sp.size
    + "x" + string sp.radius + "x" + string sp.temperature

let linesTofile path (lines: string seq) =
     use writer = File.CreateText path
     lines |> Seq.iter (fun line -> writer.WriteLine line)
```

```
let particleNameToColor name possibleNames =
    match possibleNames |> List.findIndex (fun n -> n = name) with
    | 3 -> "0"
    | 2 -> "1"
    | 1 -> "5"
    | _ -> "10"

let round (x:float) = System.Math.Round(x, 1)

let coordToString coord =
    "" + string (round coord.x) + " " + string (round coord.y) + " " + string (round coord.z)

let coordsToStringSeq coords name names t (minx,miny,minz,maxx,maxy,maxz) =
    coords |> List.fold (fun ls c ->
                         (coordToString c) + " " +
                         (particleNameToColor name names) + " " +
                         string (round t) + " " +
                         string (round minx) + " " +
                         string (round miny) + " " +
                         string (round minz) + " " +
                         string (round maxx) + " " +
                         string (round maxy) + " " +
                         string (round maxz)
                         ::ls) []
    |> List.toSeq

let amtToStringSeq amt name names t (maxX:float) (maxY:float) =
    [(particleNameToColor name names) + " " + string(amt) + " " + string (round t) + " " +
        string (int maxX) + " " + string (int maxY)]
    |> List.toSeq

let saveSnapshots snaps resultSize simulations space maxX maxY boundaries =
    let names = snaps |> List.fold (fun ls (resultSPN,_) -> (resultSPN.marking |> Map.fold (
        fun ps p _ -> p::ps) [])::ls)[]
    let parms = (paramsToString resultSize simulations space)
    let directory = "C:\\Users\\Joachim\\Dropbox\\Speciale\\data\\"
    let rec genPaths ps amtls i m =
        if i = m then (ps,amtls)
        else let newps = (String.concat "" [ directory; "particles"; parms; "-"; (string i); "
            .txt"])::ps
             let newamtls = (String.concat "" [ directory; "amount"; parms; "-"; (string i); ".
                 txt"])::amtls
             genPaths newps newamtls (i-1) m
    let (particlePaths,amtPaths) = genPaths [] [] snaps.Length 0
    let rec genParticlesAndAmtSeq names snaps ps amtls =
        match (names,snaps) with
        | (n::ns,(s,t)::ss) -> let newps = (s.marking |> Map.fold (fun ls p tks -> Seq.append
            (coordsToStringSeq (Array.toList tks) p n t boundaries) ls) (List.toSeq []))::ps
                               let newamtls = (s.marking |> Map.fold (fun ls p tks -> Seq.append
                                     (amtToStringSeq (tks.Length) p n t maxX maxY) ls) (List.
                                    toSeq []))::amtls
                               genParticlesAndAmtSeq ns ss newps newamtls
        | _ -> (ps,amtls)
    let (particlesStrings,amtStrings) = genParticlesAndAmtSeq names snaps [] []
```

```
printf "paths: %A\n strings: %A\n" amtPaths amtStrings
let toDoParticles = List.zip particlePaths particlesStrings
toDoParticles |> List.iter (fun (path,lines) -> linesTofile path lines)
let toDoAmounts = List.zip amtPaths amtStrings
toDoAmounts |> List.iter (fun (path,lines) -> linesTofile path lines)
```

Appendix R

Plotter.fs

```
module Plotter

open FnuPlot
open System
open Simulator
open Statistics
open System.Drawing

let gp = new GnuPlot()

let plotdata (simulations:SimulationResult<_>) (titles:string list) (colors:Color list)
     showminmax =
    let (datasets,minmaxsets) = fst(fst simulations)
    let axis = if showminmax then
                   let maxes = List.collect (fun (max,min) -> [max]) minmaxsets
                   globalMaxofData maxes
               else globalMaxofData datasets
    let maxX = fst axis
    let maxY = snd axis
    let points = if showminmax then
                     datasets@(List.collect(fun (min,max) -> [max]@[min])minmaxsets)
                 else datasets
    if not((List.length points) = (List.length colors)) || not((List.length points) = (List.
         length titles)) then
        failwith "please provide proper amount of colors and titles."
    let combined = List.zip points titles |> List.zip colors
    let lines =
        combined |> List.fold (fun ls (color,(data,title))
```

```
                            -> (Series.Lines( data, title = title,lineColor = color))::ls)
                                []
    gp.Set(style = Style(fill = Solid), range = Range.[0.0 .. maxX, -1.0 .. maxY*1.2])
    lines |> gp.Plot
```

Appendix S

viz.m

```
size = int2str(4000);
simulations = int2str(1);
unitLength = int2str(100);
radius = num2str(4);
temperature = int2str(298);
snapshots = 420;
particlepath = 'C:\Users\Joachim\Dropbox\Speciale\data\particles';
amtpath = 'C:\Users\Joachim\Dropbox\Speciale\data\amount';
speedUp = 500;

mov(1:snapshots) = struct('cdata', [],'colormap', []);
set(gca,'nextplot','replacechildren');
amountdata = [];
timedata = [];
typedata = [];
FigHandle = figure;
set(FigHandle, 'Position', [100, 100, 1200, 500]);
for n = 1:snapshots
    particlefilename = strcat(particlepath,size,'x',simulations,'x',unitLength,'x',radius,'x',
        temperature,'-',int2str(n),'.txt');
    [x,y,z,t,deltaT,minx,miny,minz,maxx,maxy,maxz] = importer2(particlefilename);
    amtfilename = strcat(amtpath,size,'x',simulations,'x',unitLength,'x',radius,'x',
        temperature,'-',int2str(n),'.txt');
    [type,amt,deltaT,maxX,maxY] = importer3(amtfilename);
    pause(deltaT/speedUp);
    subplot(1,2,1)
    scatter3(x,y,z,[],t);
    axis ([minx(1) maxx(1) miny(1) maxy(1) minz(1) maxz(1)]);
```

```
27      ti = strcat('temperature: ',temperature,' kelvin | radius: ',radius,' units | frame: ',
            int2str(n),'/',int2str(snapshots));
28      title ({ti});
29      subplot(1,2,2)
30      amountdata = [amountdata amt];
31      timedata = [timedata deltaT];
32      typedata = [typedata type];
33      plot(amountdata(4, :),'LineWidth',2);
34      axis ([0 maxX(1) 0 (maxY(1)*1.5)]);
35      title ({'concentration over time'});
36      mov(n) = getframe(gcf);
37  end
38
39  %aniName = strcat('animation-',speed,'x',radius,'x',snapshots,'.avi');
40  movie2avi(mov, 'ANIMATION.avi', 'compression', 'None');
```

Bibliography

[ABRS12] Omar Awile, Ferit Bueyuekkececi, Sylvain Reboux, and Ivo F. Sbalzarini. Fast neighbor lists for adaptive-resolution particle simulations. *COMPUTER PHYSICS COMMUNICATIONS*, 183(5):1073–1081, 2012.

[AS13] Tadashi Ando and Jeffrey Skolnick. On the importance of hydrodynamic interactions in lipid membrane formation. *Biophysical Journal, Biophys. J*, 104(1):96–105, 2013.

[BFR08] J. p. Banâtre, P. Fradet, and Y. Radenac. Principles of chemical programming. 2008.

[Bio] Biocyc.org. Escherichia coli k-12 substr. mg1655 laci dna-binding transcriptional repressor. `http://biocyc.org/ECOLI/NEW-IMAGE?type=ENZYME&object=PD00763`.

[dHCKMK13] Pablo de Heras Ciechomski, Michael Klann, Robin Mange, and Heinz Koeppl. From biochemical reaction networks to 3d dynamics in the cell: The zigcell3d modeling, simulation and visualisation framework. *Biovis 2013 - Ieee Symposium on Biological Data Visualization 2013, Proceedings, Biovis - Ieee Symp. Biol. Data Vis., Proc*, pages 41–48, 2013.

[DLL+15] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17:1–19, 2015.

[DRF12] Marie Durand, Bruno Raffin, and Francois Faure. A packed memory array to keep moving particles sorted. *Vriphys 2012 - 9th Workshop on Virtual Reality Interactions and Physical Simulations, Vriphys - Workshop Virtual Real. Interact. Phys. Simul*, pages 69–77, 2012.

[Gil77] DT Gillespie. Exact stochastic simulation of coupled chemical-reactions. *JOURNAL OF PHYSICAL CHEMISTRY*, 81(25):2340–2361, 1977.

[GW09] Tihamer Geyer and Uwe Winter. An o(n-2) approximation for hydrodynamic interactions in brownian dynamics simulations. *JOURNAL OF CHEMICAL PHYSICS*, 130(11):–, 2009.

[HBH+10] Michael Hucka, Frank T. Bergmann, Stefan Hoops, Sarah M. Keating, Sven Sahle, James C. Schaff, Lucian P. Smith, and Darren J. Wilkinson. The systems biology markup language (sbml): Language specification for level 3 version 1 core. 2010.

[IMR86] LANG I, SCHOLZ M, and PETERS R. Molecular mobility and nucleocytoplasmic flux in hepatoma cells. *Journal of Cell Biology*, 102(4):1183–1190, 1986.

[jav] Jigcell sbml parser. `http://jigcell.cs.vt.edu/jigcell/docs/SBML/index.html`. (Visited on 07/08/2015).

[JHZG07] Dongdong Jia, Jonathan Hamilton, Lenu M. Zaman, and Anura Goonewardene. The time, size, viscosity, and temperature dependence of the brownian motion of polystyrene microspheres. *AMERICAN JOURNAL OF PHYSICS*, 75(2):111–115, 2007.

[KC10] Ahmad S. Khalil and James J. Collins. Synthetic biology: applications come of age. *NATURE REVIEWS GENETICS*, 11(5):367–379, 2010.

[Kha11] Mohd Ehmer Khan. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications, Int. J. Softw. Eng. Appl*, 5(3):1–14, 2011.

[Kit71] C. Kittel. Thermal physics. *AMERICAN JOURNAL OF PHYSICS*, 39(7):847, 1971.

[KJKC15] Albert J. Keung, J. Keith Joung, Ahmad S. Khalil, and James J. Collins. Chromatin regulation at the frontier of synthetic biology. *NATURE REVIEWS GENETICS*, 16(3):159–171, 2015.

[Kub90] HE Kubitschek. Cell-volume increase in escherichia-coli after shifts to richer media. *JOURNAL OF BACTERIOLOGY*, 172(1):94–101, 1990.

[LB14] Sune Mølgaard Laursen and Jakob Jakobsen Boysen. A modelling framework for synthetic biology, 2014.

[Lea] Learn.Genetics. Cell size and scale. `http://learn.genetics.utah.edu/content/cells/scale/`.

[lib] Software/libsbml - sbml.caltech.edu. `http://sbml.org/Software/libSBML`. (Visited on 07/08/2015).

[MAB11] Wolfgang Marwan Mary Ann Blatke, Monika Heiner. *Tutorial Petri Nets in Systems Biology*. 2011.

[MBTK84] A. M. Mastro, M. A. Babich, W. D. Taylor, and A. D. Keith. Diffusion of a small molecule in the cytoplasm of mammalian cells. *Proceedings of the National Academy of Sciences of the United States of America*, 81(11):3414–3418, 1984.

[MF13] Constantinos. Mavroidis and Antoine. Ferreira. *Nanorobotics : current approaches and techniques*. Springer, 2013.

[MHML14] Oded Maler, Ádám M. Halász, Ouri Maler, and Olivier Lebeltel. Exploring the dynamics of mass action systems. 2014.

[MP11] Peter Mörters and Yuval Peres. Brownian motion. 2011.

[MTN11] Peter J Mohr, Barry N Taylor, and David B Newell. The 2010 codata recommended values of the fundamental physical constants. *Web version*, 6(2), 2011.

[NH14] Doktors Der Naturwissenschaften and Holger Hermanns. Stochastic satisfiability modulo theories: A symbolic technique for the analysis of probabilistic hybrid systems dissertation zur erlangung des grades eines. 2014.

[NNH99] Flemming. Nielson, Hanne Riis. Nielson, and Chris. Hankin. *Principles of program analysis*. Springer-Verlag, 1999.

[Pat06] Ron Patton. *Software testing*. Sams Pub., 2006.

[pyt] libsbml python api: Libsbml features at a glance. `http://sbml.org/Special/Software/libSBML/docs/python-api/libsbml-features.html`. (Visited on 07/08/2015).

[RM14] Nicholas Roehner and Chris J. Myers. A methodology to annotate systems biology markup language models with the synthetic biology open language. *ACS SYNTHETIC BIOLOGY*, 3(2):57–66, 2014.

[S0:] New synthetic biology technique boosts microbial production of diesel fuel | berkeley lab. `http://newscenter.lbl.gov/2012/03/26/dsrs-boosts-microbial-production-of-diesel-fuel/`.

[Wid14] Alexander R Widdel. Synthetic biology: Secure digital storage, dna-based computation and the organic computer. In *Conference*, page 55, 2014.