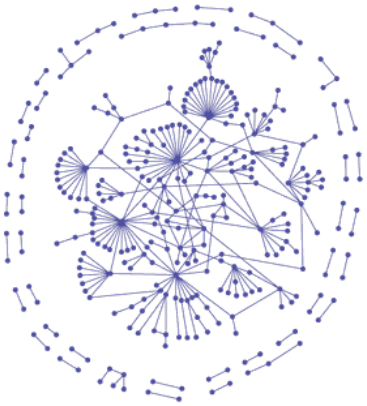
Rosemary Francis

Generating definitions of cell cycles in π-calculus from mathematical models



Part II Computer Science Tripos

Newnham College

May 16, 2005

Title page image courtesy of Sergei Maslov of Brookhaven National Laboratory, NY

Proforma

Name:	**Rosemary Francis**
College:	**Newnham College**
Project Title:	**Generating definition of cell cycles in π-calculus from mathematical models**
Examination:	**Part II Computer Science Tripos, 2005**
Word Count:	**9187**
Project Originator:	Rosemary Francis
Supervisor:	Dr P. Liò

Original Aims of the Project

To design a process by which mathematical models of biochemical systems may be translated into stochastic π-calculus. This required the design of an input language to suitably describe the mathematical models. The purpose of which was to write a translator to automate the translation process. The translator should be easy to use by biochemists and computer scientists and require no in-depth knowledge of either subject. The purpose of such a translator being to encourage the use of stochastic π-calculus in the modelling and simulation of biochemical systems.

Work Completed

As well as designing and implementing the translator I was able to contribute to the study of stochastic π-calculus in the context of biological modelling languages. I was able to address some of the problems with π-calculus and suggest possible solutions within the language.

Special Difficulties

none

Declaration

I, Rosemary Francis of Newnham College, being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Acknowledgements

First of all I have thank Pietro for supervising my project and giving up so much time to help me. I also owe a lot to Andrew and Luca for answering my questions on MAPK and SPiM. Thanks also to Kate and Katy for giving me so much encouragement. Thank you to Dad for all your spell checking abilities.

Contents

Chapter 1

Introduction

The motivation for this project stems from the rapidly developing area of Bioinformatics and the increased need for computer science to meet the computational requirements of modern genetic analysis. This is a journey into concurrency semantics and the ability of process calculi to represent the real world accurately.

Figure 1: Protein map of a yeast cell[1]

Recent advances in genetics have provided a good deal of information about the genetic make-up of cells, but they still do not tell us how the cells work. Figure 1 illustrates the complexity of the information available.

Having mapped the genome of many organisms, we can say what is in the cell, but not what would happen if changes were made to the DNA. Simulation is the way in which the dynamics of a cell cycle may be understood.

Currently, simulation of biological systems is primitive, mostly involving building

[1]Graph showing the interactions between protein within a yeast cell cycle.
Image courtesy of Jeff Kennedy Associates

mathematical models of the system via a series of rate equations. These can be plotted and stable solutions found but, as concentration[2] levels are modelled as continuous values, the models are only indicative of results from systems with very high concentrations where individual molecules have little effect. In reality, there are many systems sensitive to individual molecules.

1.1 Why Stochastic π-calculus?

Stochastic π-calculus is a process calculus designed to model non-deterministic systems. It has the facility of parallel execution, and models communication between individual processes. It is ideally suited to handling mathematical models of rate reactions, as the concept of rate and concurrent action is implicit in the language.

Non-deterministic modelling of biochemical systems allows for far greater accuracy than mathematical models. This because it is possible to model systems sensitive to small changes in integer concentration accurately. Using a non-deterministic model, each time a system is simulated there will be slightly different results. This is what you would expect of real life experiments, since the reaction rate represents only the probability of the reactants meeting and a reaction happening. Mathematical models give us only the average value.

Using π-calculus allows you to run simulations of complex experiments, vastly cutting down on the time and expense needed for running real experiments.

1.2 Previous work

System modelling languages similar to stochastic π-calculus have been around for a long time, used in the study of concurrency theory. The laguage was origionally developed by Robin Milner[3] for modelling systems such as the telephone network. It is fairly recently that bioinformatics researchers have become interested in the study of π-calculus. The stochastic π-calculus was developed by Corrado Priami[4] for the purpose of modelling biological systems. Stochastic π-calculus encorporates the notion of assigning probabilities to events needed for stochasti simulation.

Various π-calculus simulators have been written. In particular I have been us-

[2]It is assumed that the systems are closed with fixed volume, so concentration is proportional to the total number of molecules

[3]Communicating and Mobile Systems : The Pi-Calculus[R. Milner, 1999]

[4]Stochastic pi-Calculus [C. Priami, 1995]

ing SPiM[5]. One reason for choosing SPiM over other simulators is that SPiM was written specifically with biochemical uses in mind. It is well known in the field of formal language specification for bioinformatics and is used by those at the forefront of current research[6]. It is newer and more up to date with current developements than its main competitor, BioSPI[7].

1.3 The project

This project describes the design and implementation of a genetic network to stochastic π-calculus translator, SPiT. During the course of formalising a system-independent modelling standard needed for automatic translation I have also explored how useful stochastic π-calculus is for modelling biochemical systems.

> Currently not enough people have the inter-disciplinary skills needed to describe biochemical systems in π-calculus. An automatic translator would bridge the gap between concurrency theory and the study of genetic networks.

By improving the way in which systems are represented, and making the simulation techniques more accessible, I hope to encourage more wide spread use of π-calculus simulation.

1.4 Applications

One application which is of particular interest at the moment is cancer gene detection. After mapping the genome of a particular cell group and its mutations by collecting data on the local effect of each gene, you can use microarray analysis to do a 'spot the difference' test of a cancerous cell compared to a healthy one. You can then model the healthy cell with a different combination of the differences each time until you find out which combination of genes causes the cancerous behaviour.

Future work hopes to incorporate this kind of simulation into virus design in the battle against viruses such as HIV.

[5]Stochastic π-calculus Machine [Phillips 2004]
[6]In particular Luca Cardelli's group, Microsoft Research, Cambridge
[7]The BioSPI Project: http://www.wisdom.weizmann.ac.il/ biospi/

The application I will focus on is the simulation of the cell cycle, in particular the modelling of the signalling mechanisms. Currently the genes have been mapped, but little is understood about the cycle as a whole. The importance of each gene/protein is so far unknown.

Chapter 2

Preparation

The requirements of the translator were relatively straightforward. It should translate any valid biochemical system into a valid pi-calculus system with the same behaviour. The specifications for the language developed follow in the chapters to come. In order not to constrain the tool, invalid systems should be allowed and the behaviour documented where appropriate. I defined a valid system to be any that can be represented by a list of chemical reactions. An invalid system in a mathematical model is one which cannot possibly correspond to s system of chemical reactions.

It should be easy to use, and require no knowledge of π-calculus or SPiM. Ideally the program would be used by biochemists and therefore should assume very little knowledge of computer science.

Although the translator is tailored towards use alongside the SPiM simulator, the translator should be designed such that it is a simple task to adapt it for other simulators and to update it should the syntax of SPiM alter during routine updates.

All of these goals were taken into account during the design of the input language and the translation algorithm and will be discussed in more detail in the following chapters.

2.1 Language choice

SPiM is written in an object oriented functional programming language called OCAML. In order to allow later integration of the translator into SPiM, the translator was also written in OCAML.

OCAML was used to write SPiM because it allows a clear mapping from the

formal specification of the simulator onto the actual code. It is much easier to implement a formal specification in a functional language because of the highly mathematical syntax. This helps to reinforce the soundness and completeness of SPiM, by having the implementation close to the specification, and has allowed formal proof of its correctness.

The flexibility of OCAML data structures facilitates the manipulation of information in the translator. This is why functional programming languages are often used for compilers and translators. OCAML was therefore the obvious choice for implementation of the translator.

2.2 Planning

During the planning stage I was careful to factor in time to learn how to use the various languages and tools necessary to complete the given tasks. Having reviewed well-known design models and their relevant merits, I designed a flexible plan with short-term achievable goals, which allowed me to constantly review my progress and re-assess my plan.

Design pattern

Implementation of my project lent itself well to two parallel paths of execution. The tasks were not carried out entirely independently, but two tasks were on the go at any one time. This not only meant that I remained motivated as I could switch tasks at any time, but also that I was never dependent on others returning information quickly. The module structure was chosen after studying compiler design patterns. I identified key points in the translation process, and encapsulated each of them within a module.

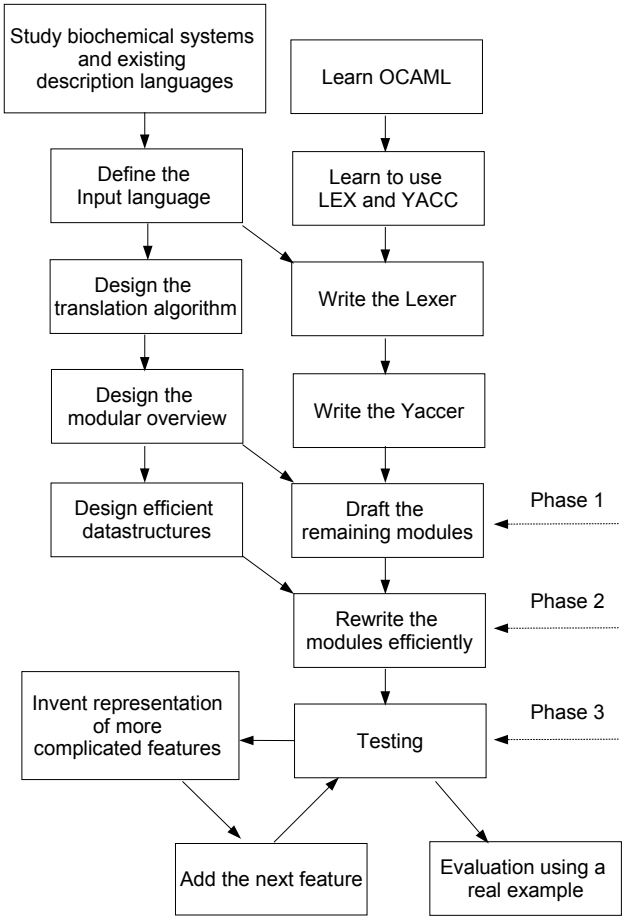


Figure 2: Flow diagram of work undertaken

The main body of the translator was designed in three phases. The first was an abstract implementation written for the purpose of getting to grips with the OCAML language and the features available. This first version was never designed to be used, and no care was taken to write efficient code, but it was a good foundation for a more refined version to emerge. This preliminary version was where much of the module structure design emerged.

Once the capabilities of the language had been explored, the second version of the translator was written with appropriate data structures and efficient recursion. This was still a skeletal implementation designed only to translate the simplest biochemical system, but it led the way for the final phase when more complicated features of the semantics were implemented in stages. This method was similar to the spiral model[1] often used in software engineering.

The features that were added one by one are listed below. All will be discussed

[1]Boehm 1988

in greater detail in later chapters.

- Constant rate declaration
- Substance concentration declaration
- Duplicate products
- Decay reactions
- Reactions with more than two reactants
- Error checking and warnings
- Sample rates and simulation length

2.3 Testing

A lot of potential problems were avoided by the language choice. The OCAML strong implicit type system means that where the modules did not fit together properly this was picked up by the compiler when the module signatures were compiled. Once the modules compiled it was straightforward to integrate them, as a lot of the hard work had been done with separate compilation. Where the data structures matched, so did the data.

I was then able to use each module to test the next. By printing a representation of the data structure passed to and from each module, I was able to identify problems within each module as I went. Once one stage was manipulating the data correctly, I was able to use it to test the next. This was a lot more productive than trying to hand code sample data for testing individual modules, as even the most simple example is nested quite deeply within data structures.

The modular structure was tested and re-evaluated during the first stage, and the intermediate data structures adjusted appropriately for the OCAML language. Once phase two was complete and tested, I was able to implement and test various features of the biochemical systems individually in a recursive design pattern. Testing was carried out between each change so that a working version was maintained throughout.

Simulation

As well as testing the translator on contrived examples, designed to look for possible bugs, I also used some previously unseen data[2] to provide a final benchmark of its performance. This was chosen from a yeast cell network. It had the advantage of being unseen from my point of view as well as including hand written π-calculus for SPiM showing typical simulation results.

[2]C-Y. Huang and J. Ferrell, Ultrasensitivity in the mitogen-activated protein cascade, 1996

Chapter 3

Implementation

In this chapter is described the design and implementation of SPiT[1] and the translation algorithm. No special knowledge of π-calculus or biological systems is assumed. Readers not familiar with π-calculus and biological systems can consult the glossary for a reminder of important terms where neccessary.

3.1 The Input Language

The input language was designed with particular regard for ease of use. Biochemists are fond of complex notation particular to their field, so care had to be taken to ensure that the notation I adopted was accepted by biochemists generally.

Notation

Biochemical systems can be represented in a variety of different forms. Most will be familiar with the chemical reaction notation with the reactants on the left and the products on the right. In this notation, a double arrow can be used to denote a two way reaction. The diagrammatic form is useful to give an overview of the system. Its appearance gives rise to the term 'genetic network'. The terms genetic network and protein network are often used interchangeably as genes are merely design patterns for the proteins that control them.

[1] Stochastic Pi-calculus Tanslator

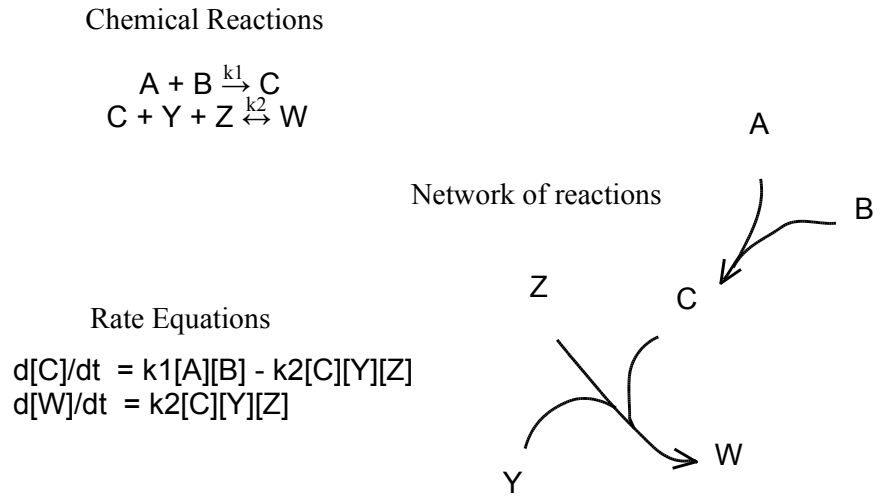


Figure 3: Three ways to represent the same system

It is the system of rate equations that is normally described as a mathematical model and it is this that is used as a basis for the input language. The model consists of the system of rate equations characterising the rate of change of concentration of a given substance. In chemistry [A] is used to denote the concentration of substance A and the constants k_i denote finite rates of reaction dependent on a number of factors.

A system of rate equations can be derived simply from a set of chemical reactions. I chose to focus on using rate equations because most systems of interest are already represented as such, and others are easily translated from systems of chemical reactions.

The rate of a reaction depends on the rate constant and the concentrations of the reactants. The rate equation for a particular substance can be built up from terms pertaining to the separate reactions involving that substance as a product or reactant. Each term, in relation to the concentration of a substance, is positive or negative depending on whether that term increases or decreases the concentration.

This should become clear when you match up the terms in the equations in the system above with the relevant reactions. For example, the rate equation of **C** has a positive and a negative term because **C** is a reactant in one reaction and a product in the other.

Creating a standard

Many attempts have been made to produce a standard for representing these equations. One reasonable successful effort resulted in SBML[2]. This is a versatile and complex language for representing a variety of biological systems and concepts in a computer-readable form. The language has the capacity to represent a great deal of information and for that reason even very simple examples become bloated. I chose to imitate the format of SBML rather than implement the language itself, to reduce the amount of irrelevant information that had to be included in the model for the translator. The input language had to be easy to use. The language I have devised is simple and intuitive by comparison with SBML's verbose syntax and lengthy documentation.

The Grammar

The input file for the translator has three main sections. The first declares all the constants and their rates. All constants have to be declared explicitly here. The second section contains information about the initial concentrations of the substances. If the initial concentration is zero then it is perfectly valid to omit that substance from this section. A warning will be printed, however, in case a typing error has led to an undeclared substance.

The following grammar is defined using BNF notation. For those not familiar with this notation it is recommended to view a few samples of input files, to be found in later sections and the appendix.

[2]Systems Biology Markup Language www.sbml.org

```
input ::= constant_def_list %% substance_def_list %% equation_list

constant_def_list ::= constant_def
                    | constant_def_list constant_def

substance_def_list ::= substance_def
                     | substance_def_list substance_def

equation_list ::= equation
                | equation_list equation

constant_def ::= constant = float ;
substance_def ::= substance : int ;

constant ::= ident
substance ::= ident

ident ::= [a-z, A-Z][a-z, A-Z, 0-9, _, ']*

equation ::= d[substance]/dt = term_list;

term_list ::= term
            | - term
            | term_list - term
            | term_list + term

term ::= constant substance_list
       | constant * int substance_list
       | int * constant substance_list
       | int constant substance_list

substance_list ::= [substance]
                 | [substance] substance_list
```

3.2 The Stochastic π-calculus

The understanding of the subtleties of my project depends on an understanding of the stochastic π-calculus as implemented by SPiM. Included below is a brief overview of the elements of the language generated by the translator, followed by a few examples. I am not attempting to explain the relationship between the π-calculus and the biochemistry at this stage; later sections of the dissertation deal with this.

Syntax

Process ::=	`new` *name*:float`:<>` *Process*	Finite rate channel restriction
|	`new` *name* `:<>` Process	Infinite rate channel restriction
|	`new` *name* `:<`*type*`>` *Process*	Typed rate channel restriction
|	*Process* `|` *Process*	Parallel execution
|	`!`*Channel Process*	Replicated channel
|	*Summation*	Choice
|	`if` *Boolean* `then` *Process* `else` *Process*	Conditional Execution
|	`(` *Process* `)`	Parenthesised process

Summation ::=	`()`	Idle process
|	*Channel*`;` *Process* `+` *Summation*	Choice

Channel ::=	*name* `<>`	Output channel
|	*name* `()`	Input channel

Parallel execution

Processes execute in parallel with the following semantics [3] . Processes existing in parallel, independent from each other with the usual concurrency issues.

$$\frac{p \rightarrow p'}{p \| q \rightarrow p' \| q} \quad \frac{q \rightarrow q'}{p \| q \rightarrow p \| q'}$$

Choice

A process can consist of several terms in a summation. These represent the possible different behaviours of the process. In SPiM the behaviour is chosen *non-deterministically* according to the Gillespie algorithm[4]. This algorithm allows systems to be represented as a form of random walk governed by the probabilities associated with the reaction rate constants.

$$\frac{p \rightarrow p'}{p + q \rightarrow p'} \quad \frac{q \rightarrow q'}{p + q \rightarrow q'}$$

Restriction, Channels and Communication

π-calculus processes 'reduce' from one state to another. All reduction is via channel communication. Output channels communicate with input channels of the same name with the following semantics:

[3] The semantic notation reads: if p reduces to p' then $p \parallel q$ can reduce to $p' \parallel q$

[4] Exact stochastic simulation of coupled chemical reactions [Gillespie, 1977]

$$\overline{a();p \parallel a\langle\rangle;q \quad \rightarrow \quad p \parallel q}$$

Input channels and output channels are differentiated by different types of brackets and complement each other. In the absence of any information being passed, it does not matter which way the communication goes, only that every communication is between one input and one output channel.

All channels are restricted. Restriction is a form of scoping. Channels may exist only within a process. For the purposes of this project all channels may be considered global.

All channels must be declared explicitly with a `new` *name*:float rate:`<>` expression. The rate determines the probability of a communication occurring. Rates can be finite or infinite. The higher the channel rate the more likely it is to communicate. Infinite rate channel communications have priority above all others. During the communication, values may be passed if the channel is typed. In modelling biochemical reactions this facility is only used for initialisation and all typed channels have infinite rate.

It is this notion of channel rate and relative probability of a communication that stochastic π-calculus differs from classic π-calculus.

Replicated action

In order to model a system, processes need to be instantiated and terminated. A process may terminate by becoming the idle process. Processes are instantiated with a replicated action. These, too, have to be paired with an appropriate input or output.

$$\overline{!a();p \parallel a\langle\rangle;q \quad \rightarrow \quad p \parallel !a();p \parallel q}$$

Graphical representation

Although there is a standard for the graphical representation of π-calculus, it is almost as hard to understand as the calculus itself. I have used a non-standard representation here, to try to make the main features of the language clearer to understand.

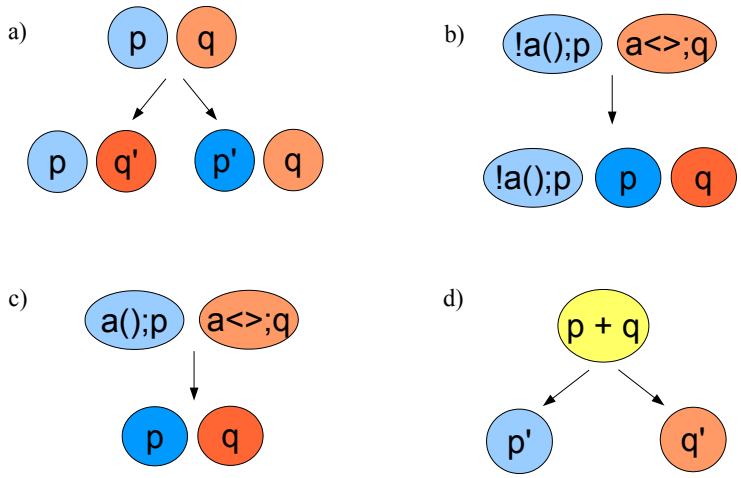


Figure 4: Graphical representation of (a)parallel execution, (b)replicated action, (c)communication and (d)choice

Part(a) of Figure 4 shows two processes p and q either becoming p and q' or p' and q.

Part(b) shows the instantiation of a process via a replicated channel.

Part(c) shows the change of state following a communication.

Part(d) shows a non deterministic choice between two possible behaviours within a process.

3.3 Modelling biochemical reactions

Previous work

Much work has been done by key people[5] in the field of using systems calculi for the modelling of genetic networks. As each model is built up slowly by hand this leads to a very good model, but relies heavily on specialist knowledge of the system in question.

In every system there are some physical substances, the total concentrations of which will remain constant. The substances merely switch between bound and

[5]Cardelli, Milner, Priami

unbound states as they interact. In the past it has been normal to represent each of these substances as a separate process cycling between various states.

```
!H();
    new e:10.0:<>
    share<e>; (h_bound<>;() + e<>; H<>)
| !Cl(); share(e); e(); Cl<>
```

The snippet of π-calculus above illustrates my point. It represents the reaction $H + Cl \rightarrow HCl$ where the hydrogen and chlorine atoms share an electron as they bond to become HCl. The electron is shared via the `share` channel where upon the hydrogen atom can take on either a bound state, `h_bound`, or communicate with the `Cl` process via the `e` channel to return the system to its original state. The `H` process cycles between a bound and unbound state indefinitely.

This is an excellent representation of this system but it requires intimate knowledge of the substances in question. For example you would have to know that these two bond covalently instead of simply becoming two ions in a solution. Also, with more than one reaction it soon becomes a nightmare to represent.

With more than one product you would have to make a decision about which reactant became which product. There is also no notion of a reactant dividing into two products.

A fresh approach

I propose a far more general approach to the translation. I begin by describing a simple framework upon which I started to make more important decisions about the details of the translation.

Each substance, be it a product or a reactant, is modelled as a process in parallel composition with every other reactant or product. You cannot have parallel composition within a substance; this feature of the pi-calculus is used solely to allow the substances to coexist in the system. For example in the simple reaction $Na + Cl \rightarrow NaCl$ you would start with Na and Cl in parallel: $Na|Cl$.

The choices represent different binding possibilities. A reaction is represented as a communication between two processes. The channel via which they communicate is named after the reaction rate constant and has the associated reaction rate. π-calculus is designed so that the reaction rate corresponds to the rate of the channel via which the reaction happens. In this way each reaction is an individual event in its own right. The only consequences are the products of the reaction.

The products are separate processes, independent of the reactants. After a reaction has been simulated via a finite channel communication, the products are instantiated via a series of (infinite rate) replicated channel communications. In this way the reactants cease to exist after the reaction and leave no evidence of having existed. To get back to the previous state, another reaction must occur with the previous products as reactants and the previous reactants as products.

Example

The following example is the simplest you can represent. We are modelling the reaction between a sodium (Na) and a chlorine (Cl) atom which becomes sodium chloride (NaCl) with rate 100.0. I am assuming for the purposes of example that this is an irreversible reaction. SPiM allows us to omit the idle process from the syntax in order to abbreviate the code.

```
new ionize:100.0:<>
new Na:<>
new Cl:<>
new NaCl:<>
new Init:<int>

new NaCl_stable:0.0:<>

( !Na(); ionize<>
| !Cl(); ionize(); NaCl<>
| !NaCl(); NaCl_stable<>
| !Init(n); if n > 0 then (Na<> | Cl<> | Init<n-1>)
| Init<1>
)
```

In the code above there are two sections. The first is the channel declaration. The first channel `ionize` represents the reaction and therefore has a finite rate. The next two are used for instantiating the sodium, chlorine and sodium chloride molecules. The fifth channel declaration is a special channel used for instantiating the system. The following three declarations are idle processes needed to get the correct data from SPiM; they can be ignored in this example.

We are initialising the system with an atom of sodium and an atom of chlorine by using an `Init` channel. This is standard practice when writing π-calculus models.

`Init<>` has infinite rate and so will communicate with `!Init()` which will then loop, instantiating the sodium and chlorine with the replicated inputs `!Na()` and `!Cl()`.

```
( !Na(); ionize<>
| !Cl(); ionize(); NaCl<>
| !NaCl(); NaCl_stable<>
| ionize<>
| ionize(); NaCl<>
)
```

The `ionize<>` and `ionize(); NaCl<>` processes represent an atom of sodium and an atom of chlorine respectively. They can communicate via the `ionize` channel. After a communication the two processes will reduce to an idle process and a process with an `NaCl` output channel.

```
( !Na(); ionize<>
| !Cl(); ionize(); NaCl<>
| !NaCl(); NaCl_stable<>
| ()
| NaCl<>
)
```

The `NaCl<>` channel communicates with the replicated input channel to reduce the system to one containing only an `NaCl_stable<>` process. Processes beginning with replicated input actions are never included as active in the system.

```
( !Na(); ionize<>
| !Cl(); ionize(); NaCl<>
| !NaCl(); NaCl_stable<>
| NaCl_stable<>
)
```

3.4 Developing the model

Apart from the simple framework described above, little attempt has previously been made to standardise the representation of biochemical systems in stochastic π-calculus. In the next few sections I will explain and justify the process by which I did this in order to automate the translation process.

Decay reactions

Decay reactions are unlike any others. They occur when one substance decays into one or more others. These are common in solutions or where the chemical bonds are weak. They present a problem because a reaction is occurring without any interaction of substances.

Other versions of π-calculus are better suited to the modelling of certain features of biochemical systems. When studying the work of others in the field I came across some which had accommodated a τ action. This is an action that can be performed at any time with no communication. It is unfortunate that this had not been combined with the stochastic nature of the π-calculus that I have studied.

$$\overline{\tau.p \rightarrow p}$$

One way to get around this is to have a dummy process. Initially I intended to have a dummy process created with each substance capable of a decay reaction. So the reaction $A \rightarrow^k B$ would look like this...

```
...
!A(); A_dummy<> (k<>;() + ... ) |
!A_dummy(); k(); B<>;() |
!B(); (...) |
...
```

This is not by any means a neat solution and does not represent the low level behaviour of the substances accurately, but it would give the correct data when simulated. `A` and `A_dummy` would communicate via `k` at the correct rate, then both would become idle processes and leave a new active `B`. The creation of `A_dummy` makes no difference and is instantaneous if it has infinite rate. Should `A` react differently and not decay at all then `A_dummy` would simply continue to exist in the system as it is.

A neater solution is to dispense with the dummy reaction and use the reaction constant itself. This means that instantiation of the dummy process does not have to be dealt with. The input end of the channel stands alone as a replicated channel which then instantiates the results of the decay.

```
...
!A(); (k<>;() + ... ) |
!k(); B<>; () |
!B(); (...) |
...
```

Multiple reactants

So far we have looked only at reactions with two or fewer reactants. Reactions with more than two are extremely rare, but nonetheless they are defined as valid

chemical reactions and must be catered for.

Reactions with more than two reactants cause a problem because only two processes may communicate at any one time. Channels have to be chained in order to involve more processes. Realistic rates may be maintained by having the first link in the chain as a finite channel then the others following with infinite rates. This will maintain the correct stochastic behaviour whilst preventing a race condition where the third reactant is 'swiped' away by another reaction during a chained one. A race condition cannot happen because infinite channels have priority over finite ones where a suitable input is available. Below is the representation of the reaction $A + B + C \rightarrow^k D$. After `A` and `B` communicate (via `k1`), `A` becomes the idle process and `B` and `C` go on to communicate (via `k2`) before `C` instantiates `D`.

```
!A(); (k1<>; () + ...) |
!B(); (k1(); k2<>; () + ...) |
!C(); (k2<>; D<>; () + ...) |
!D(); (...) |
```

Race conditions and Deadlock

Race conditions can occur when there is no third reactant available. There is no way of ensuring that there is an available third reactant before the reaction starts. If there isn't then the first two may be locked together indefinitely. Consider the system containing the following two reactions.

$$A + B + C \rightarrow D$$
$$X + Y + C \rightarrow Z$$

If the system contains no `C` atoms but plenty of the other atoms, then either reaction may occur many times and build up a system full of deadlocked substances. They will stay like this until a C atom is created in the system. There would then be a race condition between the two reactions over the possession of the C atom.

This is similar to what happens in real chemical reactions. Most three way reactions are really of the form below.

$$A + B \rightleftharpoons AB^* + C \rightarrow D$$

Generally, in three way reactions, the reactants combine in two stages. Where there is no third reactant available, the bound state of the first two reactants decays into its original substances. Thus, A and B oscillate very quickly between stable and unstable states and occasionally react with C during their unstable

state. It would make sense to try and imitate this by allowing the first two reactants to undo their reaction; all attempts to do this have led to an inconsistent system as we cannot be sure of the individual reaction rates.

```
!A(); (k1<>; () + ...) |
!B(); (k1(); (k2<>; () + A<>;B<>;()) + ...) |
!C(); (k2<>; D<>; () + ...) |
!D(); (...) |
```

In the example above, after a communication via channel `k1` has occurred, process `B` can either communicate with `C` via `k2` or instantiate a new `A` then `B` to return to the original state. Since `k2` and `A` both have infinite reaction rates the Gillespie algorithm[6] may choose either one with equal probability. So, half the time, the reaction will be undone even when there is a `C` atom available. This effectively halves the probability of a completed reaction. Doubling the reaction rate would solve this problem in the long run, but each successful reaction would effectively take twice as long. This would have huge repercussions on the system as a whole and would not be a realistic representation of the real life system at all.

The only way to solve the problem of deadlock within the system is to encourage users to enter a three way reaction as three separate reactions with an unstable substance:

$$A + B \rightarrow AB^* \qquad AB^* \rightarrow A + B \qquad AB^* + C \rightarrow D$$

This is only needed where the number of `C` atoms in the system may reach zero. Otherwise the π-calculus generated faithfully represents of the real system. The likely concentrations of C will either be obvious from the initial concentrations or can be deduced from one simulation.

Information redundancy

The representation of systems in the form of differential rate-equations has great redundancy of data. Each term of the equation represents the contribution of a particular reaction to the change in a substance's volume. To recap, the rate equations for the reaction $A + B \rightarrow^k C + D$ are:

$$\begin{array}{ll} \frac{d[C]}{dt} = k[A][B] & \frac{d[D]}{dt} = k[A][B] \\ \frac{d[A]}{dt} = -k[A][B] & \frac{d[B]}{dt} = -k[A][B] \end{array}$$

The equations for A and B are redundant as they contain no new information. In a larger system, with more reactions, A and B may be products of another

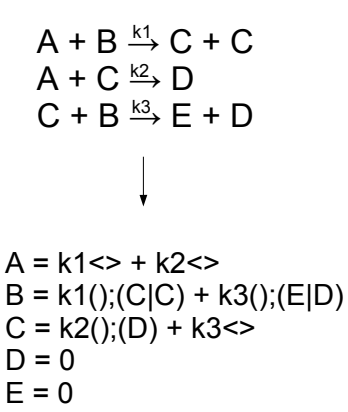
[6] Exact stochastic simulation of coupled chemical reactions [Gillespie, 1977]

equation and so their equations may be needed to represent the system.

The choice is now to either to enforce the existence of negative terms as part of the valid system or to ignore them completely. The goal of my project was to represent all valid systems correctly. By ignoring the negative terms, I can also translate invalid systems into valid systems in π-calculus. By allowing translation of systems with only positive terms specified, the readability of the input file can be greatly improved. A quick glance at the larger example in the appendix should convince you of the benefits of simplifying the inputs needed.

3.5 Translation

The Translation algorithm was developed from techniques used to make hand translation easier. The idea of using a table to represent the information was already in existence[7] and is extremely practical for hand translation.



	k1	k2	k3
A	-1	-1	
B	-1		1
C	2	-1	-1
D		1	
E			1

Figure 5: Table used for hand translating systems into π calculus.

This idea is to have a column for each reaction, identified by its reaction rate constant, and a row for each substance. Entries are integer values representing the role a particular substance plays in the corresponding reaction. Negative values imply that the substance is a reactant and is 'used up' on the left hand side of the reaction. Positive values imply that the substance is a product and is 'created' by the reaction. Follow the entries for substance **C** in the table above. In the first reaction two molecules are created whereas, in each of the second and third, one molecule is destroyed. Blank entries are where the substance is not

[7]Reference: ...

involved in that particular reaction.

A valuable feature of the table is that it is equally easy to create it from a set of rate equations as from a set of chemical reactions.

Hand translation

Manual translation is performed by traversing the table vertically, building up expressions in π-calculus for each substance in turn. Assuming every reaction has exactly two reactants, we traverse the table as follows.

- For each negative entry we add an output action to the substance's process.
- For each positive entry we add that substance to the list of substances being created in parallel after a communication.
- Care must be taken to pair up input and output channels.

This will produce a set of processes that can be combined in parallel to make a complete system in π-calculus without too much problem. However this method is inefficient both in time and space[8] and is not suitable for systems that contain reactions with more or fewer than two reactants.

Machine translation

The algorithm implemented by the translator is similar to that described above, but has linear complexity rather than quadratic complexity. A different structure is used along with various other techniques for coping with systems other than the most basic.

Decay reactions

Reactions with only one reactant are not a problem. When no corresponding negative entry is found to make up a pair of reactants, the channel is added as a `decay` channel type. This is dealt with by the module that turns the information into a string to be fed into SPiM.

Chained reactions

Since reactions with more than two reactants need chained actions, a different strategy was needed to traverse the table. An array of constants and table entry lists was used to conserve space. The 'table' therefore has to be traversed reaction

[8] $O(n^2)$ since the number of constants and substances is approximately the same

by reaction rather than substance by substance - by column rather than by row. Each column in the table is represented as a list which is divided into positive and negative terms. Actions are then `inputs`, `outputs` or `inouts`. An `inout` is an inout channel followed by an output. This means that the translator can cope with arbitrarily long chains of communications.

3.6 SPiT

There are five main modules of the translator, and three others containing important data structures and their associated update functions. The module signatures are listed in the index, but here is a description of the function of each module and its data structures in diagrammatic form:

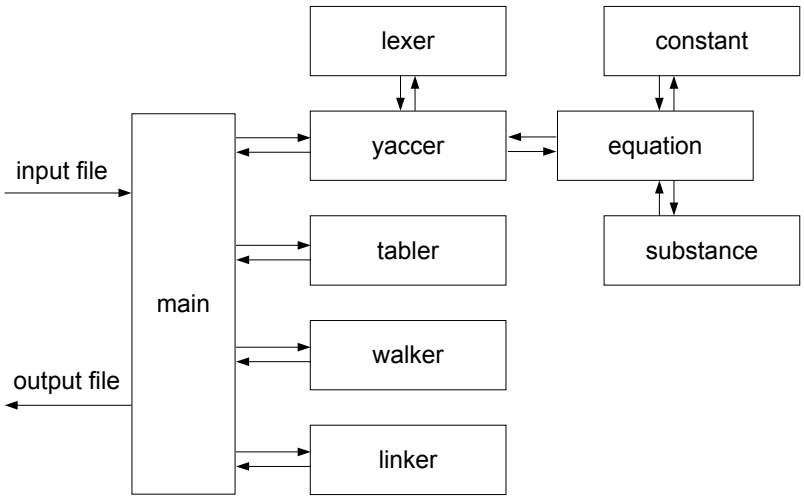


Figure 6: Modular structure of the translator

The Lexer

The lexer was generated by a program called OCAMLLEX, inspired by the C equivalent 'lex'. The semantics and usage is similar, but with a more OCAML appropriate syntax. The token details are listed in the appendix. The module simply takes an input stream and returns a token each time the lexer function is called.

The Yaccer

As the name suggests this is a parser generated by OCAMLYACC, inspired by the C equivalent 'yacc'[9]. It receives the tokens from the lexer and runs them through a finite state machine to identify the different areas of syntax and interpret them accordingly.

The input is a declaration of data, not an executable set of instructions, and so there is no abstract syntax tree. Instead the parser returns a list of equations, list of substances and a list of constants. Each equation is defined according to the equation module, with each occurring substance or constant checked for validity and referenced via its index.

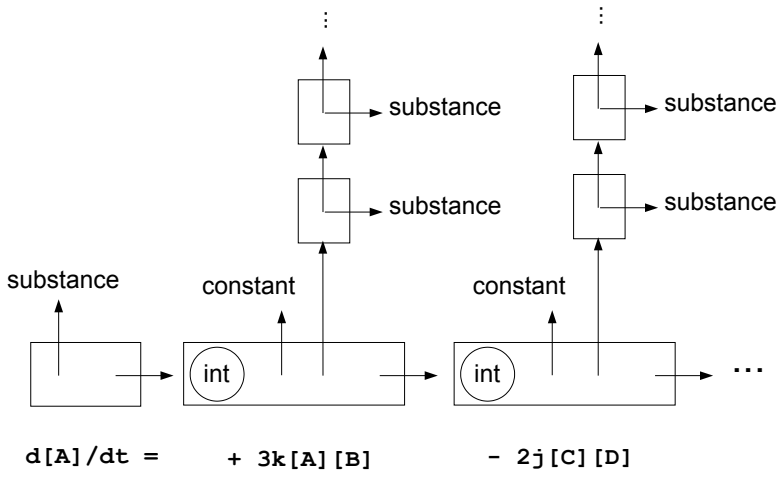


Figure 7: Equation data structure

As each new substance is encountered it is assigned an index which later becomes the index into an array to store corresponding table entries. It is also added to the list of substances. This is not an ML list, but rather a binary tree. Before the substance can be added the tree must be checked for previous occurrences. Substance names are usually very similar and will often be confused. Overloaded use is forbidden and is identified quickly. The tree ensures fast[10] search and update of the substance list by ordering it by a hash of the name. The hash is computed and stored at the nodes with the substances to avoid recompilation costs. The constants are identified and stored almost identically. Below is an illustration of the binary tree data structure and the equation list. Using a hash

[9] Yet Another Compiler Compiler

[10] O($log\ n$) - standard binary tree update and lookup

of the name prevents the pathological case[11] of alphabetically ordered substances and constants.

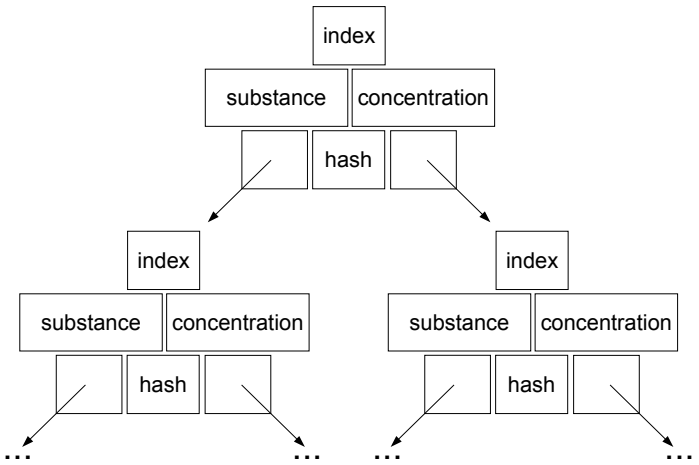


Figure 8: Binary tree used to store constants and substances

The Tabler

The tabler represents the restructuring of the data from equations into a structure similar to that of the table needed for the translation. Although a table is ideal for hand translation, the majority of the entries will be left blank and will not only waste space, but also waste time reading them. A more efficient design was used.

The tabler flattens both binary trees into an array of substances and an array of constants. It then walks the list of equations and forms a list of '(substance index, table entry): int * int' tuples for each constant. The substance index provides a direct link into the relevant array entry for that substance. The table entry indicates the relationship that substance has in the reaction corresponding to that constant[12].

The Walker

The walker walks the table and produces an expression in pseudo π-calculus for each substance. It returns an array of processes. Each process consists of a

[11]The worst case is a preordered data set, which would result in a linear structure rather than a balanced tree

[12]The table entry is positive if the substance is a product and negative otherwise. The absolute value indicates how many molecules of that substance is involved

summation detailing the behaviour of the substance in the corresponding substance array entry. Below is a snippet of code which illustrates what the array of processes (or summations) contains. For each channel there is the possibility of listing the results of the reaction in the form of an `int list`. The integers correspond to the indexes of the substances that are produced in the reaction.

```
type channel = Output of string
    | Input of string
    | Inout of string * string
    | Dumout of string;;

type summation = Idle
| Choice of channel * int list * summation;;
```

The linker

The linker links the actions of the processes together, flattens them into a string, and sandwiches them together with constant and substance declarations and initiation code.

This module is probably the simplest module. As much processing of information is done as possible in advance of this stage in order to keep it that way. This module is the only part of the translator which is simulator specific. To adapt the program for a different simulator with a similar syntax, this is the only module that would need to be re-written. In this way the translator is as versatile as possible, fulfilling the initial specification.

Complexity

Apart from the yaccer module, all other modules are linear in complexity. The majority of the work is done in building the binary trees of substances and constants, without which the other modules may well be quadratic in complexity. For each substance and for each constant there is a one-off cost of updating the tree when the substance or constant is found. This is O($log\ n$) given that the substance or constant is the n^{th}. The cost of forming the trees is therefore the sum of the cost of each update.

$$O(log\ C!S!) = \sum_{n=0}^{S} O(log\ s) + \sum_{n=0}^{C} O(log\ c)$$

where S and C are the total numbers of substances and constants

Most substances will have a corresponding rate equation, and will have fewer than ten terms referring to substances and constants, so we can approximate the

cost of processing the equations to O($S\ log\ S + S\ log\ C$). We know that n^n grows more quickly than $n!$ so if we assume that the number of substances and constants are going to be about the same, the complexity of the program can be reduced to O($n\ log\ n$) which is a very acceptable value.

Coding style

OCAML is a language with a modular system as well as the facility for object-oriented programming. During the first stage of implementation I explored various usages of the language, and eventually decided on a style similar to that used for SPiM.

I saw no need to take advantage of the capacity for object-oriented programming as the modular system provides more than adequate encapsulation. Experiments made with object-oriented models quickly became verbose, and so instead I used the syntactically more beautiful CAML[13] datatypes.

I took full advantage of OCAML's less 'functional' features. Most of the data structures are centred around arrays and mutable references. The language developed to describe the mathematical models is extremely declarative in nature, as is the stochastic π-calculus, and so did not lend itself to a purely functional way of processing.

I decided to continue with OCAML rather than move over to CAML or a completely different language, despite the lack of English language documentation for the newer language, in order to maintain the possibility of complete integration with SPiM at a later date.

[13]OCAML was developed from the non-object oriented member of the ML family, CAML

Chapter 4

Evaluation

4.1 Testing

The evolutionary model of design used in the preparation stages of the project allowed various different testing methods to be employed. Testing and evaluation was an ongoing process throughout. This ensured that as the project evolved I had always a working system to go from and that any difficulties were encountered and dealt with early on.

Algorithm testing

During the design of the translation algorithm it was necessary to constantly road-test possible methods by hand. By writing a practice draft I was able not only to improve on the way data was structured but also rethink which module was responsible for which part of the translation.

Syntax Testing

In order to make informed decisions about the way different features of the biochemical systems should be represented within the π-calculus, it was necessary to try and test many different ideas. The semantics of SPiM π-calculus are complicated by the stochastic nature of the language.

It was necessary to run many examples to convince myself of the equivalence of different ways of describing systems in π-calculus. I had to ensure that two systems that I deemed to be semantically equivalent did in fact give convincingly similar results and that the expected behaviour was observed.

Code Testing

The testing model devised at the preparation stage worked well. To recap, I implemented a skeleton form of each module and tested them all together. I used the output of one module to test the next, printing datastructures where necessary. I then went on to add more functionality to the translator, testing between each change.

Most errors were caused by common mistakes, like traversing an array in the wrong direction or following the left sub-tree instead of the right, and were picked up quickly.

As well as checking for errors in translating, I also tested appropriate errors and warnings. I introduced mistakes into the input file and checked that the translator produced the correct warning or error before exiting gracefully.

Simulation

Finally, it was time to see how my generated π-calculus compared to small scale system-specific examples that come with SPiM. By translating and simulating each example, I was able to see the translator in action for a variety of different systems before moving onto a much more ambitious data set.

A neat little example is included below. This is one of the code samples which comes with SPiM.

```
(* H + H <==> H2 *)
(* H = share<>, H Bound = h_bound<> *)
0.2/1000
new share:5.0:<<>>
new h_bound:0.0:<>
new H:<>
new Init:<int>

( !H();
    new e:0.05:<>
    ( share<e>; (h_bound<>;() + e<>; H<>)
    + share(e); (h_bound<>;() + e(); H<>)
    )
| !Init(n); if n > 0 then (H<> | Init<n-1>)
| Init<100>
)
```

Below is the code generated for this reaction by my translator.

```
0.2/1000
new k2:0.50:<>
new k1:5.0:<>

new H2:<>
new H:<>

new H2_idle:0.0:<>
new H_idle:0.0:<>

new Init:<int>

( !H2(); (k2<>;() + H2_idle<>)
| !H(); (k1(); (H2<>;()) + k1<>;() + H_idle<>)
| !k2(); (H<> | H<>)
| !Init(n); if n<100 then
  (if n<100 then H<> |
   Init<n+1> )
| Init<0>
)
```

The two examples, although syntactically quite different, give convincingly similar results. The variation is due to the stochastic nature of SPiM. No two simulations should be the same.

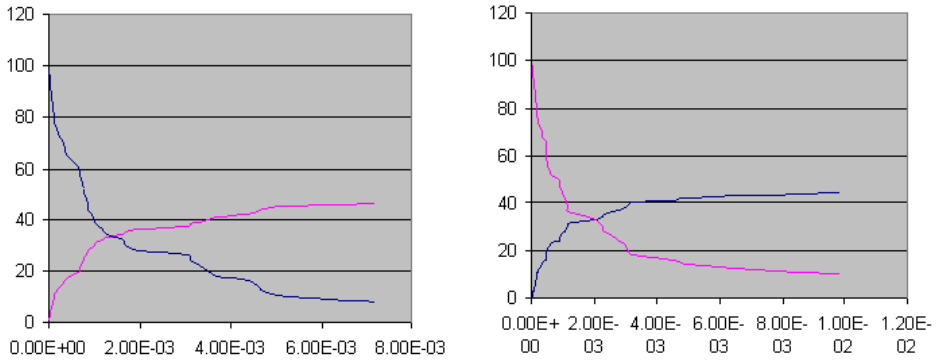


Figure 9: (left) SPiT generated (right) Hand written calculus

Simulation was a valuable part of testing. Once the translator was producing valid π-caculus, the π-caculus had to be tested to see if it represented the system correctly. In most cases the simulation results were satisfactory. Earlier experiments and careful consideration of the semantics had given a clear indication of

the behaviour of most systems.

One problem discovered during simulation was that of binding hierachy within the π-calculus. The choice operator '+' bound more tightly than expected. It was not included in the SPiM manual and I made the wrong assumptions. This misinterpretation of the syntax was picked up when I began simulating and comparing the example systems handwritten by Andrew Phillips for SPiM. Once discovered it was quickly rectified.

4.2 Mitogen-Activated Protein Cascade

The title of this project describes the ultimate goal to be the generation of cell cycle models in π-calculus from existing mathematical models. So far I have demonstrated only that it can translate small theoretical models with similar behaviour. I move now to a larger example taken from a real cell cycle.

The cell cycle is a highly regulated process. Many complex signals are needed to control the replication of the cell. One of the most significant signalling mechanisms is the MAPK cascade and it is this that I chose to model.

Why MAPK?

One of the beauties of modelling systems in SPiM is that the concentrations of the substances are discrete so that every reaction is modelled as an individual event. The MAPK cascade is extremely sensitive. It takes one input enzyme molecule to activate the three stage process which then outputs possibly hundreds of protein molecules. The responsiveness of the system can then clearly be seen. [1]

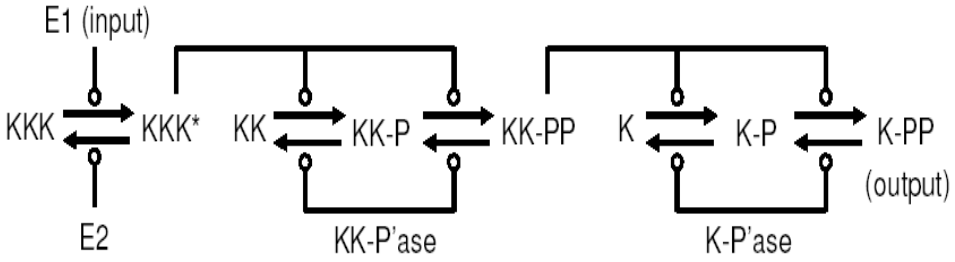


Figure 10: MAPK reactions with marked catalyst action

Work regarding this system has already been undertaken by Luca Cardelli[2]. He

[1] 'Ultra-sensitivity in the mitogen-activated protein cascade' Chi-Ying F. Huang and James E. Ferrell, 1996

[2] Microsoft Research

has hand-translated and simulated the system, although for a different version of SPiM. This gave me an excellent base to start from, as I could compare the results I got from my generated π-calculus with the results he has published online.

Results

Both my handwritten input file and the translated π-calculus can be found in the appendix. Here I include only a graphical comparison of the results.

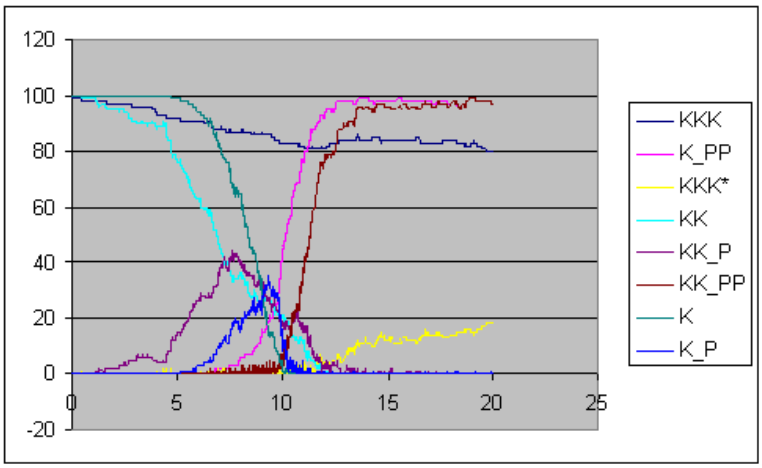


Figure 11: Results from SPiT generated π-calculus

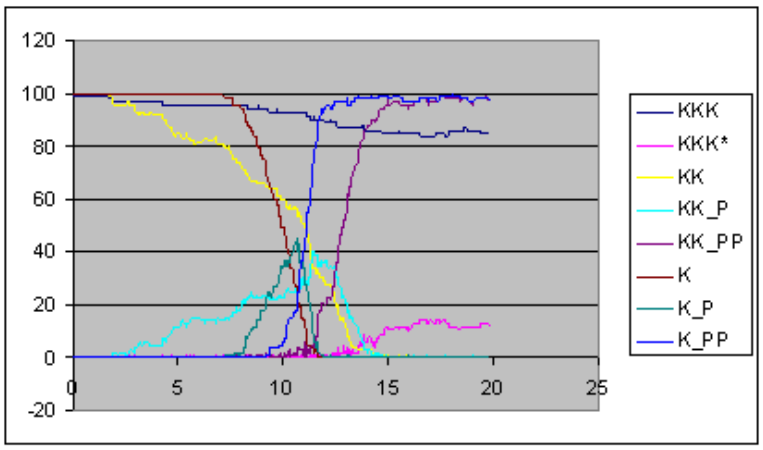


Figure 12: Results from hand translated π-calculus

Due to the stochastic nature of the systems, the graphs are not expected to look

identical. Each version was simulated multiple times to ensure that in general the graphs gave the same results. The slight differences between the two graphs are typical of the slight variations between simulations. Two sets of results from the same π-calculus model would vary similarly.

Chapter 5

Conclusion

I consider that I have made a small yet significant contribution to the study of π-calculus and the simulation of biochemical systems. At this point, I attempt to evaluate what has been achieved and justify any changes I would have made with hindsight.

5.1 How suitable is stochastic π-calculus in the modelling of genetic networks?

This is the question that I set out to explore, and am now able to answer. No language can contain everything, but π-calculus has what is needed to represent the biochemical systems to which I have applied it. I have demonstrated that any system that can be represented as a system of biochemical equations, and fits the appropriate behaviour model expected of the Gillespie algorithm, can be translated into π-calculus and simulated to obtain appropriate results. I have thus produced a practical translation algorithm and tool, but that is not to say that stochastic π-calculus was necessariy the best language to translate into.

The event-driven method of simulating reactions with distinct processes for each substance is highly suited to biochemical systems and their patterns of interaction on the whole, but there are many different type of reactions and not all fit such a uniform conception.

The introduction of a τ channel would create a cleaner syntax. By allowing processes to reduce from one state to another independently from the rest of the system, you can effectively model decay reactions directly without the use of dummy processes or other such clumsy solutions. In order to find a sensible way of representing decay reactions I had to move away from the event based model of reactions.

It is fortunate that few reactions have more than two reactants, as the representation of such reactions is complex. To introduce the possibility of deadlock into the system was undesirable, as it does not accurately represent real life systems. Ideally, three-way reactions would only occur when all three substances are available, and the reaction would be a single event. The stochastic π-calculus is not at all suited to represent such equations.

I have yet to encounter a π-calculus with the facilities to represent equations with more than two reactants, yet within the broader study of systems calculi and concurrent languages it is not hard to come across a language wherein this is possible. Petri-nets are a prime example of this capability. They are designed to be event driven, with any event able to have an indefinite number of preconditions.

It is unfortunate that many features of concurrent languages suited to representing biochemical systems do not have any means of stochastic simulation, without which there is no way of producing any realistic simulation data. I conclude that further work is required before π-calculus is ready to give a truly accurate model of biochemical systems and genetic networks.

5.2 How closely have I met the goals set by my proposal?

In the introduction of my project proposal, I said I aimed to 'create a bridge' between the knowledge of the biochemists and the knowledge of the computer scientists. I have created a comprehensive translator that will cope with a wide variety of different systems. Where stochastic π-calculus has the capability, SPiT can translate the system into it.

Initially I divided the work up into two main areas: the formal specification of an input language, and the translator itself. Both of those tasks were planned and completed within the reasonable time frame set. The second task was the more challenging.

5.3 How could this project have been improved with hindsight?

With hindsight, the first thing I would have changed is the timetable that I worked out for the project proposal. I managed to reach most of the milestones within approximately the correct time frame, but that was partially due to the underestimation of some tasks trading off against the over estimation of others.

I would have liked more time to learn the languages that I have used. I expected the task to be easier than it was; and I underestimated the difference between ML, with which I was already familiar, and OCAML.

I would still have made the same design choices. I am satisfied with the simulation results and the extent to which I have been able to demonstrate the validity of the translations performed by SPiT.

5.4 Future directions

The language of π-calculus will continue to evolve, and with it will the simulators. I hope SPiT will continue to be of some use. It should be straightforward for SPiT to evolve with future versions of SPiM. At the time of writing there is a new version of SPiM about to be made public. It is fortunate that I put such a great emphasis on abstraction and adaptivity. It is anticipated that only the last module, the linker, will need alteration to cope with the newer SPiM syntax.

I hope that, by making the simulation of genetic networks accessible, I will have encouraged more biologists to consider this as an effective analysis technique and inspire important breakthroughs in medicine.

Bibliography

R. Milner **Communicating and Mobile Systems : The Pi-Calculus** 1999

C. Priami **Stochastic pi-Calculus** 1995

C. Priami, A. Regev, E. Shapiro, W. Silverman **Application of a stochastic name-passing calculus to representation and simulation of molecular processes** 2001

M. Takane **Modeling Gene Regulatory Networks with Recurrent Neural Networks** 2002

Kutter, Neihren and Blossey **Gene Regulation in the π-calculus: Simulating Cooperativity at the Lambda Switch** 2004

A. Phillips and L. Cardelli **A Correct Abstract Machine for the Stochastic π-calculus** 2004

D. Gillespie **Exact stochastic simulation of coupled chemical reactions** 1977

C-Y. Huang and J. Ferrell **Ultrasensitivity in the mitogen-activated protein cascade** 1996

L. Cardelli **MAPK Cascade** 2005

L. Cardelli **Biological networks in Stochastic π-calculus** 2005

SPiM
http://www.doc.ic.ac.uk/ anp/spim/

BioSpi
http://www.wisdom.weizmann.ac.il/ biospi/

SBML
http://sbml.org/

Looking for Clues About How Proteins Talk to Each Other
http://www.bnl.gov/bnlweb/pubaf/pr/2002/bnlpr050202.htm

Sampling of complex networks in nature and society
http://www.jeffkennedyassociates.com:16080/connections/concept/

Glossary of terms

Genetic network A directed graph with nodes representing genes and an edge indicating the activation of one gene by another. Genes do not interact directly. They interact via the proteins they encode, so the terms genetic network and protein network are often used interchangeably.

CAML *Categorical Abstract Machine Language* A general purpose programming language that supports functional or imperative programming styles.

Mathematical model A system of rate equations describing a biological system.

OCAML *Objective Categorical Abstract Machine Language* A highly expressive language that combines object oriented programming with the ML static type system.

π-calculus Developed by Robin Milner as a formal language for describing concurrent computation processes.

Protein Network A protein network is a graph whose nodes are proteins and whose edges represent physical interactions between proteins.

Rate Equation A differential equation expressing the rate of change of concentration of a substance with respect to time.

Reaction Rate Constant A constant describing the rate of a reaction.

SPiM *Stochastic π-calculus Machine* A simulator for modelling biological systems in π-calculus [Phillips, 2004]

SPiT *Stochastic π-calculus Translator* A mathematical model to π-calculus translator for biological systems.

Stochastic π-calculus Developed by Corrado Priami to allow rates to be assigned to communications within the π-calculus.

Appendix A

SPiT files

A.1 Make file

```
ocamllex dlexer.mll
ocamlyacc dyaccer.mly
ocamlc -c constant.mli
ocamlc -c substance.mli
ocamlc -c equation.mli
ocamlc -c dyaccer.mli
ocamlc -c tabler.mli
ocamlc -c walker.mli
ocamlc -c linker.mli
ocamlc -c constant.ml
ocamlc -c substance.ml
ocamlc -c equation.ml
ocamlc -c dyaccer.ml
ocamlc -c dlexer.ml
ocamlc -c tabler.ml
ocamlc -c walker.ml
ocamlc -c linker.ml
ocamlc -c main.ml
ocamlc -o translator substance.cmo constant.cmo equation.cmo
  dlexer.cmo dyaccer.cmo tabler.cmo walker.cmo linker.cmo main.cmo
```

A.2 Signature files

The module signature files declare the public data structures, functions and exceptions.

constant.mli

```
(* name = rate *)
type constant = Constant of string * float;;

type constant_index = Constant_index of int * constant;;

type constant_tree = Node of constant_index
     * int
     * constant_tree ref
     * constant_tree ref
     | Leaf;;

(* update_constant_tree numofconsts const_id const_rate ctree *)
val update_constant_tree : int -> string -> float -> constant_tree ref -> unit;;
val get_constant_index : string -> constant_tree ref -> constant_index;;

exception InvalidConstantName;;
exception DuplicateConstantName;;
```

substance.mli

```
(* substance:concentration; *)
type substance = Substance of string * int;;

type substance_index = Substance_index of int * substance;;

type substance_tree = Node of substance_index
                              * int
      * substance_tree ref
      * substance_tree ref
| Leaf;;

(* update_sub_tree numofsubs sub_id sub_concentration stree *)
val update_substance_tree : int -> string -> int -> substance_tree ref -> unit;;
val get_substance_index : string -> substance_tree ref -> substance_index;;

exception InvalidSubstanceName;;
exception DuplicateSubstanceName;;
```

equation.mli

```
(* +/- const scaler process list *)
```

```
type term = Posterm of Constant.constant_index * int * Substance.substance_index list
  | Negterm of Constant.constant_index * int * Substance.substance_index list;;

(* d[process] = termlist  *)
type equation = Equation of Substance.substance_index * term list;;
```

tabler.mli

```
(* these are lists of (table entry, const/sub num) pairs *)

type constant_branch = Constant_branch of Constant.constant
  * (int * int) list;;

type table = Table of constant_branch array * Substance.substance array;;

(* tabler numofconst ctree numofsubs stree eqnlist *)
val tabler : int -> Constant.constant_tree ref ->
             int -> Substance.substance_tree ref ->
             Equation.equation list -> table;;
```

walker.mli

```
type action = Output of string
    | Input of string
    | Inout of string * string
    | Dumout of string;;


type summation = Idle
| Choice of action * int list * summation;;

val walker : Tabler.table -> summation array
                           * int array
                           * (string * int list) list;;
```

linker.mli

```
val linker : Walker.summation array
              -> int array
              -> Tabler.table
              -> (string * int list) list
              -> string;;
```

Appendix B

MAPK example

B.1 SPiT Input File

```
20.1

a1=1.0;
a2=1.0;
a3=1.0;
a4=1.0;
a5=1.0;
a6=1.0;
a7=1.0;
a8=1.0;
a9=1.0;
a10=1.0;

d1=1.0;
d2=1.0;
d3=1.0;
d4=1.0;
d5=1.0;
d6=1.0;
d7=1.0;
d8=1.0;
d9=1.0;
d10=1.0;

k1=1.0;
k2=1.0;
k3=1.0;
```

```
k4=1.0;
k5=1.0;
k6=1.0;
k7=1.0;
k8=1.0;
k9=1.0;
k10=1.0;

%%

KKK:100;
KK:100;
K:100;
E2:1;
E1:1;
KKP'ase:1;
KP'ase:1;

%%
d[KKP'ase] = -a4[KK_P][KKP'ase] + d4[KK_P__KKP'ase] + k4[KK_P__KKP'ase] -
             a6[KK_PP][KKP'ase] + d6[KK_PP__KKP'ase] + k6[KK_PP__KKP'ase];

d[KP'ase] = -a8[K_P][KP'ase] + d8[K_P__KP'ase] + k8[K_P__KP'ase] -
            a10[K_PP][KP'ase] + d10[K_PP__KP'ase] + k10[K_PP__KP'ase];

d[E1] = d1[KKK__E1] + k1[KKK__E1] - a1[KKK][E1];

d[E2] = d2[KKKst__E2] + k2[KKKst__E2] - a2[KKKst][E2];

d[KKK] = -a1[KKK][E1] + d1[KKK__E1] + k2[KKKst__E2];

d[KKK__E1] = a1[KKK][E1] - d1[KKK__E1] - k1[KKK__E1];

d[KKKst] = -a2[KKKst][E2] + d2[KKKst__E2] + k1[KKK__E1] + k3[KK__KKKst]
    + d3[KK__KKKst] - a3[KKKst][KK] + k5[KK_P__KKKst]
            + d5[KK_P__KKKst] - a5[KK_P][KKKst];

d[KKKst__E2] = a2[KKKst][E2] - d2[KKKst__E2] - k2[KKKst__E2];

d[KK] = -a3[KK][KKKst] + d3[KK__KKKst] + k4[KK_P__KKP'ase];

d[KK__KKKst] = a3[KK][KKKst] - d3[KK__KKKst] - k3[KK__KKKst];
```

```
d[KK_P] = -a4[KK_P][KKP'ase] + d4[KK_P__KKP'ase] + k3[KK__KKKst] +
          k6[KK_PP__KKP'ase] + d5[KK_P__KKKst] - a5[KK_P][KKKst];

d[KK_P__KKP'ase] = a4[KK_P][KKP'ase] - d4[KK_P__KKP'ase] - k4[KK_P__KKP'ase];

d[KK_P__KKKst] = a5[KK_P][KKKst] - d5[KK_P__KKKst] - k5[KK_P__KKKst];

d[KK_PP] = k5[KK_P__KKKst] - a6[KK_PP][KKP'ase] + d6[KK_PP__KKP'ase] -
           a7[KK_PP][K] + d7[K__KK_PP] + k7[K__KK_PP] + d9[K_P__KK_PP] +
           k9[K_P__KK_PP] - a9[K_P][KK_PP];

d[KK_PP__KKP'ase] = a6[KK_PP][KKP'ase] - d6[KK_PP__KKP'ase] -
    k6[KK_PP__KKP'ase];

d[K] = -a7[K][KK_PP] + d7[K__KK_PP] + k8[K_P__KP'ase];

d[K__KK_PP] = a7[K][KK_PP] - d7[K__KK_PP] - k7[K__KK_PP];

d[K_P] = k7[K__KK_PP] - a8[K_P][KP'ase] + d8[K_P__KP'ase] - a9[K_P][KK_PP] +
         d9[K_P__KK_PP] + k10[K_PP__KP'ase];

d[K_P__KP'ase] = a8[K_P][KP'ase] - d8[K_P__KP'ase] - k8[K_P__KP'ase];

d[K_P__KK_PP] = a9[K_P][KK_PP] - d9[K_P__KK_PP] - k9[K_P__KK_PP];

d[K_PP] = -a10[K_PP][KP'ase] + d10[K_PP__KP'ase] + k9[K_P__KK_PP];

d[K_PP__KP'ase] = a10[K_PP][KP'ase] - d10[K_PP__KP'ase] - k10[K_PP__KP'ase];
```

B.2 SPiT Generated π-calculus for the MAPK example

```
20.1
new k10:1.0:<>
new k9:1.0:<>
new k8:1.0:<>
new k7:1.0:<>
new k6:1.0:<>
new k5:1.0:<>
new k4:1.0:<>
```

```
new k3:1.0:<>
new k2:1.0:<>
new k1:1.0:<>
new d10:1.0:<>
new d9:1.0:<>
new d8:1.0:<>
new d7:1.0:<>
new d6:1.0:<>
new d5:1.0:<>
new d4:1.0:<>
new d3:1.0:<>
new d2:1.0:<>
new d1:1.0:<>
new a10:1.0:<>
new a9:1.0:<>
new a8:1.0:<>
new a7:1.0:<>
new a6:1.0:<>
new a5:1.0:<>
new a4:1.0:<>
new a3:1.0:<>
new a2:1.0:<>
new a1:1.0:<>

new K_P__KK_PP:<>
new K__KK_PP:<>
new KK_P__KKKst:<>
new KK__KKKst:<>
new KKKst:<>
new KKKst__E2:<>
new KKK__E1:<>
new K_PP__KP'ase:<>
new K_PP:<>
new K_P__KP'ase:<>
new K_P:<>
new KK_PP__KKP'ase:<>
new KK_PP:<>
new KK_P__KKP'ase:<>
new KK_P:<>
new KP'ase:<>
new KKP'ase:<>
new E1:<>
new E2:<>
```

```
new K:<>
new KK:<>
new KKK:<>

new Init:<int>

( !K_P__KK_PP(); (d9<>;() + k9<>;())
| !K__KK_PP(); (d7<>;() + k7<>;())
| !KK_P__KKKst(); (d5<>;() + k5<>;())
| !KK__KKKst(); (d3<>;() + k3<>;())
| !KKKst(); (a2<>;() + a3(); (KK__KKKst<>;()) + a5(); (KK_P__KKKst<>;()))
| !KKKst__E2(); (d2<>;() + k2<>;())
| !KKK__E1(); (d1<>;() + k1<>;())
| !K_PP__KP'ase(); (d10<>;() + k10<>;())
| !K_PP(); (a10<>;())
| !K_P__KP'ase(); (d8<>;() + k8<>;())
| !K_P(); (a8<>;() + a9<>;())
| !KK_PP__KKP'ase(); (d6<>;() + k6<>;())
| !KK_PP(); (a6<>;() + a7(); (K__KK_PP<>;()) + a9(); (K_P__KK_PP<>;()))
| !KK_P__KKP'ase(); (d4<>;() + k4<>;())
| !KK_P(); (a4<>;() + a5<>;())
| !KP'ase(); (a8(); (K_P__KP'ase<>;()) + a10(); (K_PP__KP'ase<>;()))
| !KKP'ase(); (a4(); (KK_P__KKP'ase<>;()) + a6(); (KK_PP__KKP'ase<>;()))
| !E1(); (a1(); (KKK__E1<>;()))
| !E2(); (a2(); (KKKst__E2<>;()))
| !K(); (a7<>;())
| !KK(); (a3<>;())
| !KKK(); (a1<>;())
| !d1(); (KKK<>;() | E1<>;())
| !d2(); (KKKst<>;() | E2<>;())
| !d3(); (KK<>;() | KKKst<>;())
| !d4(); (KK_P<>;() | KKP'ase<>;())
| !d5(); (KK_P<>;() | KKKst<>;())
| !d6(); (KK_PP<>;() | KKP'ase<>;())
| !d7(); (K<>;() | KK_PP<>;())
| !d8(); (K_P<>;() | KP'ase<>;())
| !d9(); (K_P<>;() | KK_PP<>;())
| !d10(); (K_PP<>;() | KP'ase<>;())
| !k1(); (KKKst<>;() | E1<>;())
| !k2(); (KKK<>;() | E2<>;())
| !k3(); (KK_P<>;() | KKKst<>;())
| !k4(); (KK<>;() | KKP'ase<>;())
| !k5(); (KK_PP<>;() | KKKst<>;())
```

```
| !k6(); (KK_P<>;() | KKP'ase<>;())
| !k7(); (K_P<>;() | KK_PP<>;())
| !k8(); (K<>;() | KP'ase<>;())
| !k9(); (K_PP<>;() | KK_PP<>;())
| !k10(); (K_P<>;() | KP'ase<>;())
| !Init(n); if n<100 then
  (if n<1 then KP'ase<> |
   if n<1 then KKP'ase<> |
   if n<1 then E1<> |
   if n<1 then E2<> |
   if n<100 then K<> |
   if n<100 then KK<> |
   if n<100 then KKK<> |
   Init<n+1> )
| Init<0>
)
```

Computer Science Tripos Part II Project Proposal

Generating definitions of cell cycles in π-calculus from mathematical models

Rosemary Francis, Newnham College

Originator: R. Francis

October 2004

Special Resources Required

The use of my own PC (700MHz Pentium, 256Mb RAM and 60Gb Disk). The use of the pwf.

Project Supervisor: Dr P. Lio

Director of Studies: K. Edgcombe

Project Overseers: Dr S. Holden & Dr M. Richards

Introduction

I will be working with the mathematical framework of non-linear differential equations describing the biochemical mechanisms of the cell replication cycle. During the four-stage cell cycle of many common genetic networks there are three checkpoints at which chromosomal reactions are halted and various checks are performed. These are characterised as stable solutions to the equations.

When simulated the model can yield information about mutant behaviour and other valuable insights without having the expense of a real experiment. However these mathematical models are not ideally suited to such simulation and greatly constrict the way in which the system can be analysed.

Better results can be gained by describing the system in a formal modelling language such as π-calculus. At the moment though translation has to be done by hand and is a tedious and repetitive task requiring specialised knowledge about the strain of π-calculus used and the format needed by the π-calculus simulation machine.

I aim to create a bridge between the biologists working on the cell data and the computer scientists working on the simulation software, which can be used to simulate cell mutations and cancers. This allows the development and testing of specialised genetic drugs without the costly and slow process of constructing real life experiments.

Work that has to be done

The project consists of two main piece of work:

1. A formal specification of the format of the mathematical description of the genetic networks and a method of translation into process calculus.

   This should not only provide a framework for the implementation side of the project but also give insights into the suitability of π-calculus in describing such genetic networks.

2. A system to automate the translation from the mathematical model into process calculus which can then be run directly on a π-calculus machine.

   The language I will use is called OCaml (Objective Caml). It is powerful and easy to use with many sanity checks, which eliminate many possible bugs. It is an open source implementation of Caml, a strongly-typed functional programming language from the ML family. This means it is ideal for

proving consistency in my translation method. This is an obvious extension to the project should I finish early.

I will use SPiM to simulate the case study network, a machine written by Andrew Phillips of Imperial College. It is a fast simulator of stochastic π-calculus with clearly defined program formats. This makes it an ideal target for my translation tool.

Difficulties to Overcome

π-calculus is already very well defined, but I will need to define a formal description language for the differential equations. The equations are not written by mathematicians and are an amalgamation of peculiar notations specific to biology. The method of constructing a suitable input file for the program should be easily accessible to non-computer scientists.

Having read other research papers on similar projects it is clear that there will not be one obvious method of translation, but many different ways of tackling the problem. There may not be a single way to represent the system in π-calculus. SPiM uses a sophisticated asynchronous, stochastic simulation approach and should provide interesting problems.

The main issues are that the grammar and semantics of the two description formats (the equations and the calculus) are very different. I will develop a multi-pass translator to cope with this as translating the equations into an alternative format may aid the translation process. Once I have formalised the endpoint data structures I will then work on the intermediate stages. SPiM optimisation should take place as a separate phase of the translation progress to allow easy adaptation for other simulation machines.

The project will be closely related to the areas of semantics and compiler construction as well as touching on formal specification and verification.

Starting Point

At the moment I am very new to π-calculus and to the way of describing cell cycles in the form of mathematical equations. I am also not very familiar with the language of OCaml. I will over the course of the research period of the project familiarise myself with all three languages. The greatest challenge will be at the biological end sifting the biology from the semantic information I need.

Aside from the part 1B courses I have little experience of parsers or semantics.

Backup

Although I plan to use my own PC for much of the work I will be able to use the pwf machines in the event of a failure. Backup will be on the pwf via sftp and on a flash memory stick.

Goals

My goal is to complete an implementation of the system described. Further analysis in any direction I will see as an extension. By the end I should be able to run an example genetic network description through the translator and obtain suitable data by running the target description on the simulation machine.

Work Plan

I have broken the term time into approximately two week slots. This fits well with the tasks I have to complete while allowing some flexibility during the vacation periods. I anticipate needing longer than two slots for the theoretical side of the project and implementation evaluation might run on if I decided to perform a proper consistency analysis.

Michaelmas Term

Fri 22nd Oct :	Project proposal deadline.
Mon 25th Oct - Fri 5th Nov :	Write formal specification of the equation format
Mon 8th Nov - Fri 19th Nov :	Begin design of the translation process
Mon 22nd Nov - Fri 3rd Dec :	Plan translation phases and data structures required

Lent Term

Mon 24th Jan - Fri 4th Feb :	Coding
Mon 7th Feb - Fri 18th Feb :	Module integration and testing
Mon 21st Feb - Fri 4th Mar :	Evaluation and genetic network simulation
Mon 7th Mar - Fri 17th Mar :	First draft of dissertation

Easter Term

Mon 2nd May - Thurs 19th May :	Exam revision and dissertation fine tuning
Fri 20th May :	Dissertation Deadline