Department of Computing
Imperial College London

Continuous and spatial extension of stochastic π-calculus

Final Year Project

Anton Stefanek

Marker: Dr. Maria Grazia Vigliotti
Second Marker: Dr. Jeremy Bradley

June 26, 2009

Abstract

In this project, we work towards a continuous and spatial extension of stochastic π calculus. The continuous semantics is a useful alternative to the discrete semantics and has been recently provided for other process algebras. Ability to express spatial properties of the models is an important practical extension, specially in Systems Biology.

Inspired by previous work [7][20], after showing results on process aggregation in stochastic π calculus ($\mathcal{S}\pi$) in form of multisets, we formulate and informally justify the continuous semantics. We show that this is tractable (in the sense that the set of resulting ordinary differential equations (ODEs) is finite) for the case of a subset of stochastic π calculus called *Chemical Ground Form* (CGF) defined in [7].

We attempt to tackle the problem of potentially infinite set of ODEs. We define two notions of finiteness, one allowing a direct analysis and another allowing further investigation of convergence results. We also provide an algorithm translating models in stochastic $\mathcal{S}\pi$ into CGF in case the finiteness is satisfied. We give a syntactical restriction of $\mathcal{S}\pi$ which guarantees finiteness. We intuitively and informally describe another condition on $\mathcal{S}\pi$ models guaranteeing finiteness.

We explore the relationship between the continuous and discrete semantics. We experimentally look at the effect of scaling populations of processes in various existing models.

We define a simple spatial extension of $\mathcal{S}\pi$. We bring the aggregation results to this extension and define an extended continuous semantics. We give an original example demonstrating advantages of this extension.

As an essential co-product, we develop an efficient, user friendly and portable tool implementing the above formalisms, with comparable simulation performance with the state of the art *Stochastic Pi Machine* (SPiM) simulator[32]. We also collect some of the available models in stochastic π calculus from Systems Biology, whose analysis can be enriched by the additional continuous semantics.

Acknowledgments

Firstly, I would like to thank my supervisor Dr. Maria Vigliotti for her support and enthusiasm.
I would also like to thank to Dr. Jeremy Bradley for his helpful discussions.
I would like to thank to all my friends for looking after me during the recent months.
Finally, I am grateful to my parents for their total love and support.

Contents

List of Figures

List of Algorithms

1 Introduction

Systems Biology is an interdisciplinary study field aimed towards *quantitative* understanding of biological systems. It searches for suitable abstractions that would, with the confirmation from experimental data, enhance knowledge about complex interactions within these systems.

Stochastic process algebras is a family of formalisms originating in the field of performance analysis of concurrent computer systems. Stochastic process algebras have recently been used to model different phenomena in Systems Biology. They provide a *formal description* of the systems and offer *model compositionality*, where complex systems can be expressed in terms of interacting subsystems. Also, coming from computer science, process algebras clearly separate their syntax from semantics – they can be thought of as *intermediate descriptions* from which different analyses can be carried out. Traditionally, they are used with *discrete* semantics, intuitively close to computer systems. A recent trend is in providing an additional *continuous semantics*, as has been done for example for *PEPA* in [20], *BioPEPA* and for *stochastic Concurrent Constraint Programming* in [3]. This allows a single process-algebraic description approaching the modelled problem from two different perspectives. Such feature is particularly desired in Systems Biology, where the interacting systems coexist on a wide range of temporal and spatial scales and neither the continuous nor discrete approach is universally suitable [41].

Another attractive feature of stochastic process algebras lies in the flexibility they offer to possible extensions. This again suits Systems Biology, where the modelled systems can be highly specialized and many different aspects need to be considered. One such extension is the addition of features expressing spatial properties of the modelled systems – a fundamental concept in a wide range of areas in Systems Biology, such as modelling of tissues or intra-cellular processes. There have been different attempts to bring spatial expressivity to process algebras and various specialized formalisms have been proposed, such as the BioAmbient calculus [35] or a compartmental extension of BioPEPA [11].

Stochastic π calculus is a stochastic process algebra that has been successfully applied to modelling in Systems Biology [9, 8, 38, 24]. It provides the discrete semantics, supported by a tool *Stochastic Pi Machine* (SPiM) [32] for simulating the described models.

Our aim is to extend stochastic π calculus with both the continuous semantics and spatial features, while still allowing re-use of existing models. We also aim to provide a tool that enables the additional analysis resulting from the continuous semantics to be applied to the existing models from SPiM.

1.1 Project Aim

We can summarize the main aims of the project as the following:

(1) We aim to provide a continuous semantics for stochastic π calculus. Some work has been done in this direction. Cardelli described a translation from a subset of stochastic π calculus to a system of ordinary differential equations (ODEs) via an intermediate translation through equivalent chemical equations[7]. The *continuous π calculus* is a process algebra based on π calculus aimed mainly towards the continuous semantics, with focus on evolutionary properties of biochemical pathways[25].

(2) Following the trend of PEPA and sCCP, [14],[2], the next step after defining the continuous semantics is an investigation into the relationship between the two semantics. BioPEPA presents results showing that the continuous semantics is a certain limit of the discrete; it is of interest to provide such results for stochastic π calculus.

(3) We aim to define a spatial extension of stochastic π calculus, making the framework more applicable to systems biology. As opposed to the *BioAmbient calculus*[35], we only aim to express *static* compartmental structure, with the hope of applying the continuous semantics to this extension. We require the formalism to support re-use of existing models from stochastic π calculus.

(4) To demonstrate the above concepts and to enhance existing models with the continuous analysis, we aim to develop a portable and user-friendly tool, making the framework more accessible to both biologists wishing to apply it and to computer scientists designing further extensions.

1.2 Contributions

The main contributions of this project come from tackling the above challenges.

- To arrive at a definition of the continuous semantics and also to enable efficient simulation, we describe a process aggregation method in terms of a *multiset representation.* In Chapter 3 we first restate the definition, semantics and structural congruence of stochastic π calculus (to which we refer to as to $\mathcal{S}\pi$). We show how the structural congruence can achieve process aggregation in form of multisets and thus provide an efficient simulation algorithm as well as serve as a basis for the continuous semantics.

- We define the *continuous semantics* of $\mathcal{S}\pi$. In Chapter 4 we define and give informal justification for a *direct* translation to a system of ordinary differential equations (ODEs). We show that the CGF subset of $\mathcal{S}\pi$ makes the resulting system of ODEs viable for numerical analysis. We provide an efficient algorithm to do so.

- In Chapter 5, we illustrate that the restriction operator, capable of producing new species (e.g. used to model complexation and polymerization) can cause the set of ODEs to be infinitely large. We give some basis for classification of *finiteness* of the continuous semantics. In Section 5.1, we formally describe two notions of what it means for a system of $\mathcal{S}\pi$ to produce an infinite set of ODEs. We then give a condition on the syntax of processes to guarantee the finiteness. We informally give intuition for a more general condition that will guarantee the systems to be useful for further analysis. We will describe how to translate $\mathcal{S}\pi$ models into CGF in case the finiteness is satisfied and do so for several existing models.

- We investigate the *relationship between the two semantics.* In Section 5.2 we experiment with various available models and provide observations of some properties of the relationship between the continuous and discrete semantics. We highlight the difference between these and some of the convergence properties for BioPEPA[14].

- We define an extension of $\mathcal{S}\pi$ that allows to express *static compartments with fixed volume.* In Chapter 6 we justify the main ideas and give the definition and provide continuous semantics for this extension. We give an example of an existing $\mathcal{S}\pi$ model extended with the spatial features and also give an original example from plant biology demonstrating the flexibility of this framework.

- We implement a portable *tool* written in Java programming language that provides efficient simulation (as an alternative to SPiM) of models in $\mathcal{S}\pi$ and ODE generation and solution of models in CGF, enhancing the possible analysis of existing and future models. Chapter 7 describes the design, algorithms and used technologies and suggests a proof of correctness. In the Appendix B, we provide a *collection of models* with results from both semantics when applicable, gathered from available literature.

2 Process algebras in Systems Biology

There has been an increasing interest in the application of process algebras in the modelling and analysis of biological systems. The reason for this is that there is an obvious correspondence between biological systems and concurrent systems – the species (molecules, proteins, etc.) can be seen as processes interacting and influencing each other (e.g. via chemical reactions). See [6] for examples of this correspondence in different areas of Biology. This abstraction provides various benefits:

- Process algebras provide a *formal representation* of the modelled system, thus avoiding ambiguity.
- They offer model *compositionality* – complicated systems can be defined in terms of subsystems. This is crucial in tackling an important theme in systems biology, which is to understand how the *interactions* between different components bring new functionality.
- Process algebras conveniently offer *different analysis techniques*. They can be considered as an intermediate description that leads to different mathematical formalisms.

Last but not least, stochastic process algebras offer direct, practical implementations. This fits well in the knowledge discovery cycle of Systems Biology. First, a *formal* process-algebraic model abstracting a biological system is proposed, using the available knowledge and intuition about its components, functionality and interactions with other systems. On this model, different mathematical and computational analyses can be performed (using the tool offered by the process algebra), thus providing an experiment *in-silico*. Results of this experiment can be compared with the real experimental data. This can lead to refined models (e.g. using more fine tuned parameters agreeing with the experiments) and eventually to better understanding of the biological system. Moreover, when the biological system is known to be consisting of different subsystems, the corresponding models can be composed together, avoiding additional work, and the analysis offering insight into the role different interactions and cooperations play in the functioning of the system. See Figure 2.1 for an overview.

We give an introduction to stochastic process algebras in the context of Systems Biology. We describe different mathematical formalisms which are used by stochastic process algebras, including the Gillespie algorithm for stochastic simulation, numerical methods for solving differential equations and mention some of the theory of Markov chains. We introduce stochastic process algebras and show how exactly they are attractive to Systems Biology and how they employ the above formalisms to provide analyses of the modelled systems. We will mainly concentrate on general process algebras, such as the stochastic π calculus and BioPEPA. We also briefly look at some extensions providing spatial modelling and the alternative continuous semantics.

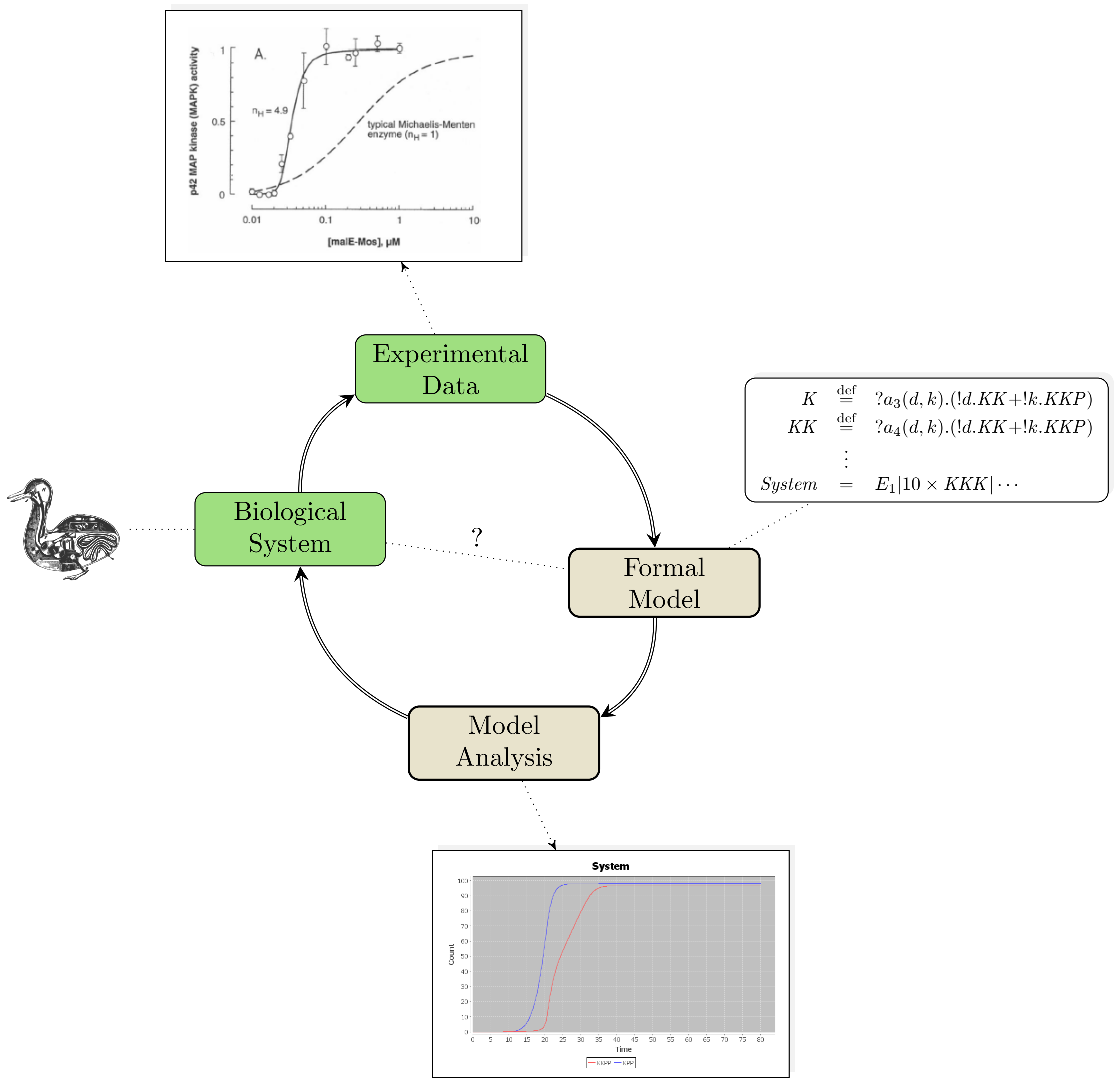

A.
$n_H = 4.9$
typical Michaelis-Menten enzyme ($n_H = 1$)
p42 MAP kinase (MAPK) activity
[malE-Mos], μM
Experimental Data
Biological System
?
Formal Model
Model Analysis
$K \overset{\text{def}}{=} ?a_3(d,k).(!d.KK+!k.KKP)$
$KK \overset{\text{def}}{=} ?a_4(d,k).(!d.KK+!k.KKP)$
$\vdots$
$System = E_1|10 \times KKK|\cdots$
System
Count
Time

Figure 2.1: The cycle of systems biology. Formal model is created using the current knowledge and intuition about the system. The model analysis is compared with the experimental data to improve the model and eventually enhance the knowledge about the biological system.

Notation. In the rest of this report, we will introduce various abstract concepts, some of which can be rather obscured by their syntax (specially the Chapter 3 and 4 can be quite "syntax heavy"). We are aware of this and will always try to give intuitive description of the underlying ideas. Moreover, we will use the margins to place markers[1] whenever a new syntactical construct is defined.

[1] *new syntax*

We will use similar style when describing the implementation, marking the important Java classes.

2.1 Mathematical background

We briefly go through various mathematical concepts that are useful in designing and implementing stochastic process algebras. We first state some well known properties of Exponential distribution which is crucial for analysis of (Markovian) stochastic process algebras. We restate a simple method for sampling variables from the Exponential and general discrete distribution, which will be useful for implementing the discrete semantics of stochastic π calculus. We define a class of stochastic processes, continuous-time Markov chains (CTMCs) that are the target formalism of many stochastic process algebras. We show how these can be simulated using the described sampling techniques. We describe the Gillespie algorithm for stochastic simulation of chemical reactions and show how it relates to the CTMCs. We recollect a numerical method for solving systems of ordinary differential equations, which will be useful when implementing computational analysis of the continuous semantics of stochastic π calculus.

2.1.1 Exponential distribution

In the following, we will use exponential random variables in various places. We state some of their important properties that can be found in standard probability literature [37, 18]. A continuous random variable X is said to have an *exponential distribution* with parameter λ if its probability density function is given by

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

Its cumulative distribution function then is

$$F(x) = \int_{-\infty}^{x} f(y)\mathrm{d}y = \begin{cases} 1 - e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

The mean of X can be derived as

$$\begin{aligned} \mathbb{E}(X) &= \int_{-\infty}^{\infty} x f(x)\mathrm{d}x \\ &= \int_{0}^{\infty} \lambda x e^{-\lambda x} \\ &= [-xe^{-\lambda x}]_0^\infty + \int_0^\infty e^{-\lambda x}\mathrm{d}x = \frac{1}{\lambda}. \end{aligned}$$

The variance is $\mathrm{Var}(X) = 1/\lambda^2$.

A random variable X is said to be *memoryless* if

$$\mathbb{P}(X > s + t | X > t) = \mathbb{P}(X > s)$$

for all $s, t \in \mathbb{R} \geq 0$. If we consider X as a lifetime of a certain object, the above states that after any arbitrary lifetime t of the object, the remaining lifetime has the same distribution as at the time 0; that is, the object does not "remember" that it has already been alive for time t.

We can simply verify that the exponential random variable X is memoryless:

$$\begin{aligned} \mathbb{P}(X > s + t) &= e^{-\lambda(s+t)} \\ &= e^{-\lambda s}e^{-\lambda t} = \mathbb{P}(X > s)\mathbb{P}(X > t). \end{aligned}$$

On the other hand, the exponential distribution is the only memoryless distribution: Let X be memoryless and let $g(x) = \mathbb{P}(X > x)$. Then we get

$$g(s + t) = g(s)g(t).$$

Letting $s = t = 1/n$, we get

$$g\left(\frac{2}{n}\right) = g\left(\frac{1}{n} + \frac{1}{n}\right) = g^2\left(\frac{1}{n}\right).$$

Similarly, for all integers m, $g(m/n) = g^m(1/n)$. Also

$$g(1) = g\left(\frac{1}{n} + \frac{1}{n} + \cdots \frac{1}{n}\right) = g^n\left(\frac{1}{n}\right)$$

and hence $g(m/n) = (g(1))^{m/n}$. As g is right continuous, we can replace m/n by any real number x. Now $g(1) = (g(1/2))^2 \geq 0$ and so $g(x) = e^{-\lambda x}$ where $\lambda = -\log(g(1))$. This will be later very useful when characterizing continuous time Markov chains.

We will use *hazard rates* when arguing that systems of chemical reactions can be represented by Markov chains.

Definition. Let X be a continuous random variable with distribution function F and density f. The *hazard rate* function $r(t)$ of X is

$$r(t) = \frac{f(t)}{1 - F(t)}.$$

One way to look at hazards is to observe, for a small δt,

$$\begin{aligned}
\mathbb{P}(t < X < t + \delta t | X > t) &= \frac{\mathbb{P}(t < X < t + \delta t,\ X > t)}{\mathbb{P}(X > t)} \\
&= \frac{\mathbb{P}(t < X < t + \delta t)}{\mathbb{P}(X > t)} \\
&\simeq \frac{f(t)\delta t}{1 - F(t)} = r(t)\delta t.
\end{aligned}$$

For X exponentially distributed with parameter λ, the hazard is

$$\begin{aligned}
r(t) &= \frac{f(t)}{1 - F(t)} \\
&= \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda.
\end{aligned}$$

Therefore the hazard of an exponential random variables is constant.

On the other hand, the hazard uniquely determines the distribution F. Integrating both sides of the hazard definition, we get

$$\begin{aligned}
\log(1 - F(t)) &= -\int_0^t r(t)\mathrm{d}t + k \\
1 - F(t) &= e^k \exp\left(-\int_0^t r(t)\mathrm{d}t\right).
\end{aligned}$$

Letting $t = 0$ shows that $k = 0$ and so

$$F(t) = 1 - \exp\left(-\int_0^t r(t)\mathrm{d}t\right).$$

In case of a constant hazard, we get the exponential distribution function (and so no other continuous distribution has constant hazard).

Consider n independent random variables $X_1, X_2, \ldots, X_n$ all exponentially distributed with respective parameters $\lambda_1, \lambda_2, \ldots, \lambda_n$. Let $Z = \min(X_1, X_2, \ldots, X_n)$. The distribution of Z can be derived as

$$\begin{aligned}
\mathbb{P}(Z > z) &= \mathbb{P}(X_i > z \text{ for all } i = 1, \ldots, n) \\
&= \prod_{i=1}^{n} \mathbb{P}(X_i > z) \\
&= \prod_{i=1}^{n} e^{-\lambda_i z} \\
&= \exp\left(-\left(\sum_{i=1}^{n} \lambda_i\right) z\right).
\end{aligned}$$

Therefore Z is exponentially distributed with parameter $\sum_{i=1}^{n} \lambda_i$. We can look at the n variables X_i as times until occurrence of n different events. The random variable Z is then the time until *any* of these events occurs. If we treat the parameters λ_i as the rates of the corresponding events, i.e. the number of events of type i occurring during a unit time, then it is expected that the rate of any event occurring occurring during unit time is the sum of all the rates.

We now look at the probability that an event occurring belongs to a chosen variable X_i. Let $W = \min(X_j,\ j \neq i)$. Then W is an exponential random variable with the rate $\lambda = \sum_{j \neq i} \lambda_j$ and so

$$\begin{aligned}
\mathbb{P}(Z = X_i) &= \mathbb{P}(X_i \leq W) \\
&= \int_0^\infty \mathbb{P}(X_i \leq W | X_i = x)\mathbb{P}(X_i = x)\mathrm{d}x \\
&= \int_0^\infty \mathbb{P}(W > x)\lambda_i e^{-\lambda_i x}\mathrm{d}x \\
&= \int_0^\infty \lambda_i e^{-(\lambda_i + \lambda)x}\mathrm{d}x \\
&= \frac{\lambda_i}{\lambda_i + \lambda} = \frac{\lambda_i}{\sum_{i=1}^n \lambda_i}.
\end{aligned}$$

The above two properties will be useful when simulating the situation of having n events with exponentially distributed delay times. We use the second property to randomly choose an event that happens next and the first one to determine the delay until this event happens.

2.1.2 Sampling from random variables

One of the output formalisms of stochastic process algebras is stochastic simulation. This requires generation of random variates. We will describe the basic inversion method that will be sufficient for the simulation algorithms described in the remainder of this report. We assume that we can generate samples from the standard uniform distribution (a common feature of most programming platforms).

Proposition 2.1. Let U be a standard uniform random variable (i.e. one taking values in $[0, 1]$) and F a monotonous distribution function. Then the random variable $X = F^{-1}(U)$ has distribution function F.

Proof. Because F is monotonous, we have for any $x \in \mathbb{R}$

$$\begin{aligned}
\mathbb{P}(X \leq x) &= \mathbb{P}(F^{-1}(U) \leq x) \\
&= \mathbb{P}(U \leq F(x)) \\
&= F(x).
\end{aligned}$$

■

Using the above proposition, we can simulate an exponentially distributed random variable with parameter λ. We have $F(x) = 1 - e^{-\lambda x}$ and so $F^{-1}(x) = -\frac{1}{\lambda}\log(1 - x)$. Therefore, if U is uniform $(0, 1)$, the random variable $-\frac{1}{\lambda}\log(1 - U)$ is exponentially distributed with parameter 1. Because $1 - U$ is also uniform $(0, 1)$, we have $-\frac{1}{\lambda}\log U$ is exponential with parameter λ.

In a similar way, we can simulate any discrete random variable taking values from a finite set. Assume U is standard uniform and X is discrete with mass function

$$f(k) = p_k$$

for $k = 1, 2, \ldots, K$ and real numbers p_i such that $\sum_{k=1}^K p_i = 1$. If we now divide the interval $[0, 1]$ into K subintervals $I_1 = [0, p_1)$, $I_2 = [p_1, p_1 + p_2)$, $\ldots$, $I_K = [\sum_{k=1}^K -1, 1]$ then clearly $\mathbb{P}(U \in I_i) = p_i = \mathbb{P}(X = i)$ for all $i = 1, \ldots, K$. Therefore we can take i such that $U \in I_i$ as a realization of X.

We just add that there are many more sophisticated methods for sampling random variables (both continuous and discrete) that can apply to a wider range of distribution and can also offer better performance of implementation. However, the inversion method will suffice for our investigation.

2.1.3 Markov chains

Building on probability theory, we can introduce the Markov chains – the target formalism for discrete semantics of stochastic process algebras.

Definition. A *stochastic process* is a family $\{X(t) : t \in T\}$ of random variables indexed by some set T. We call a stochastic process *discrete time* if $T = \{0, 1, 2, \dots\}$ and *continuous time* if $T = [0, \infty)$.

Definition. Continuous time stochastic process $\{X(t) : t \geq 0\}$ is a *continuous time Markov chain* (CTMC) if for all $s, t \geq 0$ and nonnegative integers $i, j, x(u), 0 \leq u < s$ if

$$\mathbb{P}(X(t+s) = j | X(s) = i, X(u) = x(u),\ 0 \leq u < s) = \mathbb{P}(X(t+s) = j | X(s) = i).$$

If the probability

$$\mathbb{P}(X(t+s) = j | X(s) = i)$$

is independent of s, we say the CTMC has *stationary* or *homogeneous* transition probabilities. From now on we will assume this to be the case.

The following definition is an alternative formulation of CTMCs (which can be proved equivalent to the above), more suitable for simulation.

Definition. CTMC is a stochastic process such that

(i) each time it enters a state i, the amount of time it spends in that state before making a transition into state j is exponentially distributed with parameter λ_{ij} depending only on i and j,

(ii) when it leaves state i, it enters state j with some time independent probability P_{ij}.

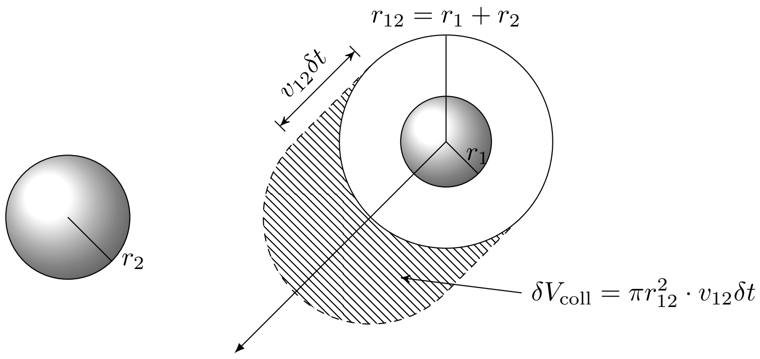
We add that there is a highly developed theory that allows more analysis of CTMCs, such as calculation of transient probability distributions or steady states. This usually relies on the state space of the CTMC to be finite. Whereas it is the case for some stochastic process algebras such as BioPEPA, we will show that models in stochastic π calculus can result in CTMCs with infinite state space. In that case, stochastic simulation will be the main technique of analysing those CTMCs. Although it does not provide precise results and is prone to error, it is efficient and with certain care can be used to gain better understanding of the underlying models.

2.1.4 Simulation of Markov chains

The second characterization of CTMCs is directly suited to simulation. The most obvious way to proceed is called the *direct method.* In each state i, there are finitely many possible transitions. The waiting times for all of these will be exponentially distributed random variables and hence the time until the first transition occurs will be exponential too, with parameter equal to the sum of the parameters of the individual waiting times. Sampling this random variable (for example using the inversion method) gives us the time until the transition occurs. Then we can get the new state j by sampling a discrete random variable with mass function $f(j) = P_{ij}$ for all possible j (by using inversion if there is finitely many of them).

2.1.5 Gillespie Algorithm

In this section we introduce the widely used simulation algorithm of Gillespie [17], who made a big contribution to bringing stochastic simulation to biochemistry. The algorithm deals with the following problem: Assume a fixed volume V containing a spatially uniform mixture of n chemical species interacting through m specified chemical reaction channels. Given the initial numbers of molecules of each species, what will the populations be at any given time?



$r_{12} = r_1 + r_2$
$v_{12}\delta t$
r_1
r_2
$\delta V_{\text{coll}} = \pi r_{12}^2 \cdot v_{12}\delta t$

Figure 2.2: The X molecule (represented by the sphere of radius r_1) moves towards the Y molecule (represented by the sphere with radius r_2, the diagram does not represent the relative positions of the two molecules), sweeping volume δV_{coll} in a short time interval δt.

Traditionally, ordinary differential equations are used to tackle this problem. Let $X_i(t)$ be the number of molecules of the i-th species at time t. Assuming that each reaction is a continuous rate process, we get the *reaction equations*

$$\begin{aligned} \mathrm{d}X_1/\mathrm{d}t &= f_1(X_1,\dots,X_n),\\ \mathrm{d}X_2/\mathrm{d}t &= f_2(X_1,\dots,X_n),\\ &\vdots\\ \mathrm{d}X_n/\mathrm{d}t &= f_n(X_1,\dots,X_n). \end{aligned}$$

It is sometimes argued that in addition to these equations usually being not analytically tractable, they also don't describe the physical basis of the problem faithfully and assume that the time evolution of a chemically reacting system is continuous and deterministic – molecular population levels can only change in discrete steps and it is not possible to account for exact positions and velocities of all the molecules in the system. Hence it is impossible to predict the system behaviour and so the time evolution is not deterministic.

Another way of looking at this problem takes a probabilistic approach, based on the assumption that the contents of the fixed volume are well stirred and hence the molecules uniformly and independently distributed over the volume V. Consider the reaction

$$X + Y \to \cdots .$$

and assume that both X and Y molecules are spheres of radii r_1 and r_2 respectively. Such reactions are the most common and more complicated ones (those involving more than two molecules) can be considered rare, see [40])

The above reaction occurs when any two X and Y molecules collide, i.e. when the distance between an X molecule and an Y molecule is less than $r_{12} = r_1 + r_2$.

We can pick an arbitrary pair of X and Y molecules and consider the speed v_{12} of the X molecule relative to the Y molecule. In the next small time interval δt, the X molecule will cover, relative to the Y molecule, a *collision volume* $\delta V_{\text{coll}} = \pi r_{12}^2 v_{12}\delta t$. Since the molecules are uniformly distributed, we can get the probability of the reaction by fixing an X molecule and calculating the probability of having a Y molecule within δV_{coll}. See Figure 2.2.

Therefore

$$\begin{aligned} \mathbb{P}(\text{the two molecules coliding in next } \delta t) &= \delta V_{\text{coll}}/V\\ &= V^{-1}\pi r_{12}^2 v_{12}\delta t\\ &= c \cdot \delta t \end{aligned}$$

for a constant c specific to the reaction. Now a probability of *any* pair of X and Y reacting is (if there are X_1 molecules of X and X_2 molecules of Y in the system) $X_1X_2c\delta t$ as there are X_1X_2

different pairs. We can see this as a hazard rate of a distribution of the time until the reaction occurs. Therefore this is exponentially distributed with parameter $X_1X_2c_i$. Generally, depending on the left hand side of the reaction equation, the number of different combinations will depend on the current population level. For each reaction i, we can denote the hazard by $h_i(\widetilde{x}, c_i)$ for a system state $\widetilde{x}$, a vector of concentrations of the individual species. Because these hazards depend only on the current state of the system, its time evolution can be regarded as a CTMC. This leads to a simulation algorithm (named the *Gillespie algorithm* in the context of biochemistry) identical to the general direct method for Markov chains. See Algorithm 1.

$h_i(\widetilde{x}, c_i)$

Algorithm 1 The Gillespie algorithm

```
1: Initialize molecule numbers in the vector $\widetilde{x}$, set time $t \leftarrow 0$
2: **repeat**
3:   Calculate $h_i(\widetilde{x}, c_i)$ for each $i$
4:   Sample the next reaction $\mu$ from discrete distribution where $i$ has a probability $h_i(\widetilde{x}, c_i)$
5:   Sample $\tau$ from exponential distribution with parameter $\sum_j h_j(\widetilde{x}, c_j)$
6:   Change the number of molecules according to the reaction $\mu$, set $t \leftarrow t + \tau$
7: **until** $t > t_{\text{stop}}$
```

2.1.6 Next reaction method

There have been several improvements to the Gillespie algorithm (and so the direct method as well), focusing on more efficient simulation. One such improvement is the algorithm by Gibson and Bruck [15], also called the *Next reaction* method, which improves the efficiency in presence of large numbers of reacting species and reaction channels. We will implement this algorithm in addition to the direct method and try to assess its suitability for stochastic π calculus.

The main idea of the algorithm is in keeping a queue of all the reactions that can happen. This may seem counter-intuitive in the first place. However, an efficient implementation of the queue allows fast retrieval and insertion of the reaction times. This is combined with an observation that not all of the times in the queue have to be updated after a reaction occurs – only the reaction involving species whose population changed have to be considered – this relationship is given by a *dependency graph* calculated prior to the execution of the algorithm. Moreover, it can be proved that the affected reaction times don't have to be re-sampled and can be scaled based on the current reaction.

Algorithm 2 The Next Reaction Method

```
1: Initialize molecule numbers, set $t \leftarrow 0$, generate dependency graph $\mathcal{G}$
2: Calculate the hazard $a_i$ for each reaction $i$
3: For each reaction $i$ generate the reaction time $\tau_i$ from an exponential distribution with parameter $a_i$

4: **repeat**
5:   Let $\mu$ be the reaction with the least $\tau_\mu$
6:   Update the numbers of molecules to reflect execution of $\mu$
7:   Sample $\tau$ from an exponential distribution with parameter $\sum_j a_j$
8:   Change the number of molecules according to $\mu$, set $t \leftarrow t + \tau$
9:   **for all** edge $(\mu, \alpha)$ in $\mathcal{G}$ **do** {update the affected reactions}
10:    update $a_\alpha$
11:    if $\alpha \neq \mu$, set $\tau_\alpha \leftarrow (a_{\alpha,\text{old}}/a_{\alpha,\text{new}})(\tau_\alpha - t) + t$
12:    if $\alpha = \mu$, generate $\rho$ from an exponential distribution with parameter $a_\mu$ and set $\tau_a \leftarrow \rho + t$
13:  **end for**
14: **until** $t > t_{\text{stop}}$
```

2.1.7 Numerical algorithms for solving systems of ODEs

The continuous semantics will lead to an initial value problem, consisting of a set of n coupled ordinary differential equations for some n,

$$\begin{aligned} \mathrm{d}y_1/\mathrm{d}t &= f_1(y_1, \dots, y_n), \\ &\vdots \\ \mathrm{d}y_n/\mathrm{d}t &= f_n(y_1, \dots, y_n) \end{aligned}$$

and initial values $y_i(0) = c_i$ for all $i = 1, \dots, n$.

In general it is intractable to find an analytical solution. In that case it still may be possible to provide an approximate numerical solution. One of the standard algorithms to do so is the *fourth order Runge-Kutta mehtod*. Starting from the initial values in $\widetilde{y}^{(0)}$, the method proceeds in determining the values of $\widetilde{y} = (y_1, \dots, y_n)$ at times that are increments of a step size h. Each $\widetilde{y}((k+1)\cdot h) = \widetilde{y}^{(k+1)}$ is determined from $\widetilde{y}^{(k)}$ considering the slope of $\widetilde{y}$ given by $f = (f_1, \dots, f_n)$, at different points between $k \cdot h$ and $(k+1) \cdot h$. See Algorithm 3 for details. We will implement this in our tool.

Algorithm 3 The fourth-order Runge-Kutta method.

```
1: set $\widetilde{y}_0$ to contain the initial values, $t_0 \leftarrow 0$
2: **while** $t_n < t_{\mathrm{stop}}$ **do**
3:     $k_1 \leftarrow f(t_n, \widetilde{y}_n)$
4:     $k_2 \leftarrow f(t_n + \frac{1}{2}h, \widetilde{y}_n + \frac{1}{2}hk_1\widetilde{1})$
5:     $k_3 \leftarrow f(t_n + \frac{1}{2}h, \widetilde{y}_n + \frac{1}{2}hk_2\widetilde{1})$
6:     $k_4 \leftarrow f(t_n + h, \widetilde{y}_n + hk_3\widetilde{1})$
7:     $y_{n+1} \leftarrow \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$
8:     $t_{n+1} \leftarrow t_n + h$, $n \leftarrow n + 1$
9: **end while**
```

2.2 Stochastic process algebras

Stochastic process algebras are based on *process algebras*. Traditionally, process algebras are formalisms to specify the behaviour of a concurrent system in a formal, modular and hierarchical way. The basic building blocks are *processes* that can perform *actions* – the basic unit of communication.

Process algebras are usually defined in an inductive fashion, giving the *basic primitives*, e.g. actions, the zero process not capable of any actions or an instance of a previously defined process, and then providing *operators* for composing these, such as summation expressing multiple capabilities, parallel composition for allowing communication, restriction or name hiding for expressing private communication etc.

The behaviour of the process algebraic models is in most cases given in terms of semantics defined on the inductive structure of processes, producing a transition system of some kind. This usually describes what happens to a process after it performs an action and when multiple processes interact and in what way they synchronize. Additionally, possible recursive behaviour or private communication can be described.

The original purpose of process algebras was to reason about the systems *qualitatively*, for example studying different notions of equivalences or validity of properties expressible in some system of logic. The development of *stochastic process algebras*, an extension of the above, was motivated by the need of analysing performance, a *quantitative* measure, of large computer and communication systems. In this setting, the process algebraic description is enriched with information about the *delay* of individual actions, usually in terms of a probability distribution. The semantics then enhances the transition system with transition distributions and thus gives rise to stochastic processes. In case the transition distributions are exponential, the resulting stochastic process is a CTMC. This allows for different kinds of analyses to be performed on the models, such as numerically solving (in case the underlying state space is finite) or simulating the CTMCs.

Due to the earlier mentioned similarities between biological systems and concurrent processes, stochastic process algebras got applied and later adapted to specific problems in Systems Biology. Through this paper, we will sometimes use biological terms in place of their process algebraic representations and vice versa, see Figure 2.3 for a small overview.

Process algebra	Biology
process	gene,protein,molecule, cell
identifier	species
communication	reaction
top-level process	mixture,solution
number of processes	population
system	model

Figure 2.3: Some correspondence between biological and process algebraic terms that will be used interchangeably in this report.

2.2.1 Stochastic π calculus

We introduce a process algebra that will serve as the basis for our further extensions. Here we only give a brief overview and postpone the details to the Chapter 3. The original π calculus was developed by Milner [27] as an extension of *CCS* and has been widely used to study concurrent and mobile computational systems. The quantitative extension – the *stochastic π calculus* has been defined in [33] and in [36] the authors argued why it can serve as a suitable abstraction for Systems Biology.

The building blocks of stochastic π calculus are processes, communicating through *channels*. Each channel has an associated *rate*, a parameter of an exponential distribution of the communication latency over the channel. Processes can perform *actions*. These can be on the channels – *output actions* $!a\langle b\rangle$ and *input actions* $?a(x)$ where a is the channel name, b the sent message and x the bound variable that gets replaced by the received message. Processes can also perform internal silent actions at a given rate, $\tau@r$.

The most basic process is the *zero process* $\mathbf{0}$, not capable of any action. A process capable of executing an action α and evolving into a new state (process) P is written as $\alpha.P$. Multiple capabilities are grouped by the summation operator $\sum$. Two processes can be put in parallel by the operator $|$ to allow communication on the channels as well as independent transitions.

The messages sent in output actions can include channel names – one of the feature distinguishing π calculus from CCS. The main reason for this is the *restriction operator* $(\text{new}\, e@r_e)$ – a way of defining a new channel e which a process can send to establish private communication. Finally, process identifiers can define processes with recursion.

The syntax of the processes of stochastic π calculus can be summarized as

$$\begin{aligned}
P ::=\ & \mathbf{0} && \text{(the empty process)}\\
:\ & \sum_{i\in I} \alpha_i.P && \text{(guarded summation)}\\
:\ & (\text{new}\, x)P && \text{(restriction)}\\
:\ & P|P && \text{(parallel composition)}\\
:\ & A\langle a\rangle && \text{(identifier instance)}
\end{aligned}$$

The semantics of stochastic π calculus is in the form of a transition system, with state corresponding to processes and labels to the rates. If a process is a parallel composition, it allows the components to perform silent actions individually. If two components can perform complementary actions (input to output and vice versa) on the same channel, they evolve together (with the rate of the used channel), possibly exchanging a message. If the sender is under a restriction and the message contains the private name, the receiver gets put under the same restriction (this is called *scope extrusion* of the restriction operator). The transition system can then be interpreted in a straightforward way as a CTMC (under the second formulation). Intuitively, since a process can evolve into a parallel composition, the CTMC can have an infinite state space, making stochastic simulation the only possible method of analysis.

We look at the stochastic π calculus in detail in Chapter 3, where we also recall *structural congruence* – an equivalence relation, that will enable aggregation of processes and thus lead to an efficient simulation algorithm of the underlying CTMC. The aggregation also shows how stochastic π calculus naturally supports the law of mass-action – the rate of reaction between two molecules is proportional to the product of their concentrations (as we argued in the description of the Gillespie algorithm).

Cardelli related a subset of stochastic π calculus, the *Chemical Ground Form* (CGF) to a subset of standard chemical equations[7], by defining a translation between the two. Since there is a standard translation of chemical equations to ordinary differential equations, this provides a continuous semantics for the CGF subset of stochastic π calculus. An interesting question is whether there is a clear *direct* translation from CGF to a set of ODEs and also whether this can be extended to the full stochastic π calculus. We will try to address this in the following chapters.

The state of the art tool for analyzing models in stochastic π calculus is the *Stochastic Pi Machine* (SPiM) developed by Phillips and Cardelli[32]. It provides efficient stochastic simulation of the models, using the Gillespie algorithm. It also provides a convenient graphical notation for the processes.

2.2.2 Continuous π calculus

A recent development is the *continuous π calculus*[25]. It is inspired by the syntax of π calculus, but generalizes several concepts. It is not restricted to input and output actions on channels. Instead, it considers *affinity networks*, a mechanism to define communication between arbitrary names, making the modelling more suitable in the context of biology – for example the names can represent sites of interaction on proteins and the networks the possible interactions. This also generalizes message exchange - when two processes communicate, both can send and receive a messages. The semantics of continuous π calculus is given in terms of real vector spaces. Although a promising formalism, it does not provide a freely available tool and does not specify how infinite systems of differential equations are treated.

2.2.3 Bio-PEPA

BioPEPA is an extension of a process algebra PEPA, built on the experience with modelling signalling pathways with PEPA [12]. In contrast to π calculus, PEPA is inspired by the process algebra CSP and offers synchronization of more than two processes. The original stochastic semantics of PEPA led to *bounded channel kinetics* where the rate of synchronization of processes is proportional to the minimum rate of the synchronized processes. This served as a limitation in the context of biology, where different kinetic laws are required. BioPEPA therefore introduces *functional rates*, allowing to express general kinetic laws and also stoichiometric coefficients. It focuses on *reagent-centric* view, where the individual processes correspond to levels of concentration of the species, as opposed to π calculus where each process corresponds to a single molecule (although the multiset representation we define in the next Chapter will move more towards this direction). Such description of the system then leads to different forms of analysis – BioPEPA offers analysis of the resulting CTMC (as it can guarantee a finite number of states), stochastic simulation, solution of the ODEs from continuous representation of the system and model checking.

The resulting CTMC is in the case of BioPEPA called *CTMC with levels.* In [14], authors show that such CTMCs also belong to the family of *density-dependent* Markov chains and so provide certain convergence properties. In particular, the authors show that a set of ODEs which is precisely the one derived from the continuous semantics, can be considered as a limit, when increasing the number of levels of concentration of the species, of the transient behaviour of the CTMC. Provided a continuous semantics is defined for the stochastic π calculus, it would interesting to relate it to this work and examine whether the set of ODEs from the continuous semantics is a limit of the CTMC in some sense.

2.3 Spatial extensions

There are numerous process algebras oriented towards expressing spatial properties. These include for example *BetaBinders* [34] or *BioAmbients* [35]. BioAmbients are a modification of Ambient calculus, which is an extension of π calculus aimed at describing *mobility.* The processes can be enclosed in compartments (the *ambients*) and are allowed to communicate as in π calculus when within the same compartment. In addition, processes are also allowed to direct the behavior of their enclosing ambients – they can move inside another ambient, merge with a neighbouring ambient, create a new ambient or dissolve an existing one. In Systems Biology, it has been argued the ambients are suitable to model mobility around membrane interactions. Although BioAmbients is a promising formalism, we believe that enhancing it with continuous semantics would be too challenging. Instead, we will concentrate on systems where the compartments are *static* – in [11], authors define an extension to BioPEPA capable of expressing static compartmental structure, with possibility of changing volume over time. They also justify the use of static compartments by the fact that those are the ones considered in models present in the literature and various specialized databases. One of our aims is to define such extension for stochastic π calculus that would also allow the re-use existing models.

2.4 Summary

In this chapter we gave an overview of the role of stochastic process algebras in Systems Biology, providing the context of the work that will follow in this report. We introduced stochastic π calculus and mentioned the reasons we find it worth of further research.

We also listed several concepts and techniques that will be useful when reasoning about and implementing our extensions.

3 Stochastic π calculus

In this chapter, we give a careful and detailed overview of the stochastic π calculus. The presentation will be based on the original paper [33] where stochastic π calculus was introduced, but modified in order to fluently lead to the implementation as well as to facilitate the further extensions. We will abbreviate the presented version of stochastic π calculus by $\mathcal{S}\pi$. $\mathcal{S}\pi$

We start by a definition of the language of $\mathcal{S}\pi$. This language gives us a way to describe the modelled system, in form of an *environment* defining the possible components and a *top-level process* representing the initial configuration. After going through several technicalities (such as substitution and alpha congruence), we define the semantics of $\mathcal{S}\pi$, in form of a transition system where the states correspond to processes and the labels to rates from channel communication or silent delays. This can lead to a CTMC and to simulation. We will argue that this is not efficient as the run time depends on population of individual processes. We therefore proceed to use the *structural congruence*, a commonly used equivalence relation in process algebras, to provide aggregation of processes in terms of *multisets*. This will then lead to an efficient simulation not dependent on the individual populations – we provide a theorem explicitly enumerating all the possible transitions in the transition system and use this theorem to give an efficient modification of the Gillespie algorithm for $\mathcal{S}\pi$.

We conclude the chapter by describing a representation that will be useful for defining the continuous semantics.

3.1 Syntax

Here we describe the language of $\mathcal{S}\pi$. We first list the different sets of words that can be used. The main elements of $\mathcal{S}\pi$ are processes. The calculus allows naming of these by *process identifiers* in order to facilitate more succinct syntax and more importantly recursion. The processes are able to communicate via *channels* - either named or anonymous dynamically created ones, both denoted by *channel names.*

Definition (Process identifiers)**.** Let $\mathcal{I}$ be a set of *process identifiers.* We usually denote these with A, B, C, etc. $\mathcal{I}$

Definition (Channels)**.** Let $\mathcal{N}$ be a countable set of *channel names.* In the following, we usually use the lower case letters a, b, c, etc. to denote these. $\mathcal{N}$

We assume functions $n\colon \mathcal{N} \to \mathbb{N}$ and $r\colon \mathcal{N} \to \mathbb{R}$ that associate channels with their *arity* and *rate.* For a channel $c \in \mathcal{N}$, we write these as n_c and r_c. n_c, r_c

Variables will serve as placeholders for receiving messages in channel communication.

$\mathcal{V}$ **Definition** (Variables)**.** Let $\mathcal{V}$ be a set of *variables*. We usually denote these with x, y, z, etc. We denote vectors of variables, i.e. elements of the set $\mathcal{V}^n$ for some $n \in \mathcal{N}$, by $\widetilde{x}, \widetilde{y}$, etc. and mixed vectors of variables and names, i.e. elements of the set $(\mathcal{V} \cup \mathcal{N})^n$ by $\widetilde{\psi}, \widetilde{\varphi}$, etc.

A basic unit of communication between two processes is an *action*. A process can either send a message (possibly containing some information) through a channel (an *output action*) or receive a message (an *input action*). Additionally, processes can perform an internal change (a *silent action*).

$!a\langle\widetilde{\psi}\rangle$ **Definition** (Actions)**.** An *output action* on a channel or a variable $a \in (\mathcal{V} \cup \mathcal{N})$ is denoted $!a$. The action can also include sending *arguments*, a vector $\widetilde{\psi} \in (\mathcal{V} \cup \mathcal{N})^n$ for some $n \in \mathbb{N}$. We write such action as $!a\langle\widetilde{\psi}\rangle$.

$?a(\widetilde{x})$ Similarly, an *input action* on a channel or a variable $a \in (\mathcal{V} \cup \mathcal{N})$ is $?a$. This action can also receive arguments, a vector $\widetilde{x} \in \mathcal{V}^n$ for some $n \in \mathcal{N}$. We write such action as $?a(\widetilde{x})$.

$\tau@r$ A *silent action* of rate $r \in \mathbb{R}$ is denoted as $\tau@r$.

$\mathcal{A}$,$\mathcal{A}_\tau$ Finally, in the following we use α to denote an action and let $\mathcal{A}$ be the set of all actions and $\mathcal{A}_\tau$ the set of all silent actions.

We can now define the structure of processes of $\mathcal{S}\pi$. The most basic process, serving as a base case for the recursive composition of processes is the *zero process*, not capable of any action. To enable actions, a process can be written as a *summation* of different alternatives – a collection of *continuations*, where each element is an action together with the further evolution of the process (another process). Two processes can be composed in parallel to enable all of their actions in addition to communication between them. A new channel name can be created in a *restriction*, ensuring that no external processes can communicate on this channel unless they are sent its name. Finally, an *instance* of a previously defined process can be reused by naming the corresponding identifier.

$\mathcal{P}$ **Definition** (Processes)**.** A set of processes $\mathcal{P}$ is defined as the minimal set such that

$\mathbf{0}$ (i) the *zero process* $\mathbf{0}$ is in $\mathcal{P}$,

$\sum_{i\in I} \alpha_i.P_i$ (ii) a *summation* $\sum_{i\in I} \alpha_i.P_i$, where I is a finite index set and α_i an action and $P_i \in \mathcal{P}$ a process for all $i \in I$, is in $\mathcal{P}$,

$P|Q$ (iii) a *parallel composition* $P|Q$, where $P, Q \in \mathcal{P}$ is in $\mathcal{P}$,

$A\langle\widetilde{\psi}\rangle$ (iv) a *process identifier instance* $A \in \mathcal{I}$ is in $\mathcal{P}$ and also its *parametrized instances* $A\langle\widetilde{\psi}\rangle$, $\widetilde{\psi} \in (\mathcal{N} \cup \mathcal{V})^n$ for some $n \in \mathbb{N}$, are in $\mathcal{P}$,

$\text{new}\, a@r_a$ (v) a *restriction* $(\text{new}\, a@r_a)P$, where $a \in \mathcal{N}$, $r_a \in \mathbb{R}$, $P \in \mathcal{P}$, is in $\mathcal{P}$.

In the following, we abbreviate $\alpha.\mathbf{0}$ as α, use the $+$ operator as an explicit version of $\sum$, that is write $\alpha_1.P_1 + \alpha_2.P_2$ for $\sum_{i\in\{1,2\}} \alpha_i.P_i$, and avoid any operator and write $\alpha.P$ in case $|I| = 1$.

We also sometimes refer to $\alpha.P$ as to a *channel continuation* if α is a channel action and *delay continuation* otherwise. The identifier instances that can be used to build processes have to be defined in an *environment* - a collection of equations which relate an identifier name (with optional parameters) to the corresponding process.

$A(\widetilde{x}) \overset{def}{=} P$ **Definition** (Environment)**.** A *defining equation* for a process identifier $A \in \mathcal{I}$ is

$$A \overset{\text{def}}{=} P$$

where $P \in \mathcal{P}$. This can be additionally parametrized as

$$A(\widetilde{x}) \overset{\text{def}}{=} P$$

n_A where $P \in \mathcal{P}$ and $\widetilde{x} \in \mathcal{V}^n$ for some n called the *arity* of A and denoted n_A.

E An *environment* E is a finite set of defining equations.

Such environments need to be well defined. In particular, all the identifiers used in the definitions have to be defined within the same environment and used with the same arity of the parameters. Additionally, none of the identifiers can *immediately produce* itself, i.e. lead to another copy of itself without any action. This restriction can be justified with the fact that in nature, processes always take some time and nothing can get produced instantly.

Definition (Valid environment)**.** Let

$$E = \left\{ \begin{array}{c} A_1(\widetilde{x}_1) \overset{\text{def}}{=} P_1 \\ \vdots \\ A_m(\widetilde{x}_m) \overset{\text{def}}{=} P_m \end{array} \right\},$$

$A_i \in \mathcal{I}$, $\widetilde{x}_i \in \mathcal{V}^{n_{A_i}}$ for $i = 1, \dots, m$ be an environment. A process P is *valid with respect to* E if

(i) the set of identifiers used by P is a subset of $\{A_1, \dots, A_m\}$,

(ii) every instance $A_i\langle\widetilde{\psi}\rangle$ of A_i is used with the right arity, that is $\widetilde{\psi} \in (\mathcal{V} \cup \mathcal{N})^{n_{A_i}}$.

An identifier A_i *immediately produces* identifier A_j if the syntax tree of P_i contains an instance of A_j which is not below a summation. Further, A_i immediately produces A_j if A_i immediately produces A_k which immediately produces A_j.

An identifier A_i is valid with respect to E if P_i has free variables (see further) precisely those in $\widetilde{x}_i$ and if it doesn't immediately produce itself.

We say that the environment E is a *valid environment* if all A_i, $i = 1, \dots, m$ are valid with respect to E.

Example 1. An environment containing the equations

$$\begin{aligned} A(x, y) &\overset{\text{def}}{=} !x\langle y\rangle, \\ B &\overset{\text{def}}{=} !c.A\langle d, e, f\rangle \end{aligned}$$

is not valid as the defining equation for B contains an instance of A with the wrong arity.

An environment containing the equations

$$\begin{aligned} A &\overset{\text{def}}{=} !a|B, \\ B &\overset{\text{def}}{=} (\text{new}\, b@1.0)(!b|C), \\ C &\overset{\text{def}}{=} !c|A \end{aligned}$$

is not valid because the identifier A can produce itself infinitely. On the other hand, the environment consisting of the equations

$$\begin{aligned} A &\overset{\text{def}}{=} !a|\tau@1.0.B, \\ B &\overset{\text{def}}{=} (\text{new}\, b@1.0)(!b|C), \\ C &\overset{\text{def}}{=} !c|A \end{aligned}$$

is valid because A no longer produces B (instantiation of a new B is guarded by the silent action $\tau@1.0$).

We will refer to a pair (S, E) of a valid environment E and a process S valid with respect to E as to a *system of stochastic π calculus* and call S a *top-level process*. This can be considered as a complete specification of a model – E describes the model and S gives the initial situation. (S, E)

Example 2. We can define a simple valid environment containing the defining equations

$$\begin{aligned} \mathit{Prey} &\overset{\text{def}}{=} \tau@r_1.(\mathit{Prey}|\mathit{Prey}) + ?\mathit{eat}, \\ \mathit{Predator} &\overset{\text{def}}{=} !\mathit{eat}.(\mathit{Predator}\ |\mathit{Predator}) + \tau@r_2 \end{aligned}$$

and a top level process

$$\mathit{System} = (\mathit{Predator}|\mathit{Predator})|\mathit{Prey}$$

with *eat* a channel in $\mathcal{N}$ with $n_{eat} = 0$, $r_{eat} = 0.01$ and $n_{Prey} = n_{Predator} = n_{System} = 0$. This can represent a simple predator–prey model, in which the prey (represented by the *Prey* process) reproduces at a rate r_1 (the silent action $\tau@r_1$ leading to two copies of the *Prey* process) unless eaten (communication on the channel *eat*) by the predator (represented by the *Predator* process), which needs the prey in order to reproduce and is otherwise dying at a certain rate (the silent action $\tau@r_2$, leading to a zero process).

We can extend this to demonstrate the use of the restriction operator and parameter passing by considering the valid environment with defining equations

$$\begin{aligned}
\mathit{HidingPlace} &\overset{\text{def}}{=} (\text{new}\, a@1.0)(!\mathit{hide}\langle a\rangle.?\mathit{discover}.!a.\mathit{HidingPlace}),\\
\mathit{Prey} &\overset{\text{def}}{=} \tau@1.(\mathit{Prey}|\mathit{Prey}) + ?\mathit{eat} + ?\mathit{hide}(x).\mathit{HiddenPrey}\langle x\rangle,\\
\mathit{HiddenPrey}(y) &\overset{\text{def}}{=} ?y.\mathit{Prey},\\
\mathit{Predator} &\overset{\text{def}}{=} !\mathit{eat}.(\mathit{Predator}\ |\mathit{Predator}) + \tau@1 + !\mathit{discover}.\mathit{Predator}
\end{aligned}$$

and the top level process

$$\mathit{System} = (\mathit{Predator}|\mathit{Predator})|((\mathit{Prey}|\mathit{Prey})|\mathit{HidingPlace})$$

where $\mathit{hide} \in \mathcal{N}$, $n_{hide} = 1$, $n_{HiddenPrey} = 1$. The modification can represent the ability of *Prey* processes to "hide" in designated places and to be later discovered by the *Predator* processes. The *HidingPlace* process uses the restriction (the "new" operator) to create a unique, private channel. This channel is then sent to the *Prey* that is going to hide in the place (by an output action on the channel *hide*). After this, the *Prey* cannot be eaten, since the only action it allows is on the private channel it received from the hiding place. When the place gets discovered by a *Predator*, the corresponding *Prey* gets notified by communication over the private channel. The fact that the channel is private makes sure the hidden *Prey* processes get paired with *HidingPlace* processes. We will precisely describe this behaviour in the semantics of $\mathcal{S}\pi$.

Note that the above example might be overcomplicated and serves only to demonstrate the use of the syntactic features.

3.2 Substitution and alpha congruence

Before defining the semantics of stochastic π calculus, certain attention has to be paid to the channel names. Mainly, we need to define how the newly generated channel names in restrictions will be treated to stay private. This also affects the behaviour of receiving a message from an input action $?a(x)$.

We will use the following definitions, all implicitly stated with respect to a valid environment E.

fn, bn **Definition** (Free and bound names)**.** Define the functions returning *free names* and *bound names* of a process, $\text{fn}\colon \mathcal{P} \to 2^{\mathcal{V}\cup\mathcal{N}}$, $\text{bn}\colon \mathcal{P} \to 2^{\mathcal{V}\cup\mathcal{N}}$ respectively, inductively as

$$\begin{array}{lcllcl}
\text{fn}(\mathbf{0}) &=& \emptyset, & \text{bn}(\mathbf{0}) &=& \emptyset\\
\text{fn}(?a(\widetilde{x}).P) &=& \{a\} \cup \text{fn}(P) \setminus \widetilde{x}, & \text{bn}(?a(\widetilde{x}).P) &=& \widetilde{x} \cup \text{bn}(P),\\
\text{fn}(!a\langle\widetilde{\psi}\rangle.P) &=& \{a\} \cup \widetilde{\psi} \cup \text{fn}(P), & \text{bn}(!a\langle\widetilde{\psi}\rangle.P) &=& \text{bn}(P),\\
\text{fn}(\tau@r.P) &=& \text{fn}(P), & \text{bn}(\tau@r.P) &=& \text{bn}(P),\\
\text{fn}(\sum_{i\in I}\alpha_i.P_i) &=& \bigcup_{i\in I}\text{fn}(\alpha_i.P_i), & \text{bn}(\sum_{i\in I}\alpha_i.P_i) &=& \bigcup_{i\in I}\text{bn}(\alpha_i.P_i),\\
\text{fn}(P|Q) &=& \text{fn}(P) \cup \text{fn}(Q), & \text{bn}(P|Q) &=& \text{bn}(P) \cup \text{bn}(Q),\\
\text{fn}((\text{new}\, a@r)P) &=& \text{fn}(P) \setminus \{a\}, & \text{bn}((\text{new}\, a@r)P) &=& \text{bn}(P) \cup \{a\},\\
\text{fn}(A\langle\widetilde{\psi}\rangle) &=& \widetilde{\psi}, & \text{bn}(A\langle\widetilde{\psi}\rangle) &=& \emptyset.
\end{array}$$

where $a \in (\mathcal{V} \cup \mathcal{N})$, $\widetilde{x} \in \mathcal{V}^n$, $\widetilde{\psi} \in (\mathcal{N} \cup \mathcal{V})^m$ for some $n, m \in \mathbb{N}$.

Note that we implicitly use the vectors $\widetilde{x}$ and $\widetilde{\psi}$ as sets when the context is clear.

Example 3. Consider the environment

$$\begin{aligned}
P(x) &\overset{\text{def}}{=} \underbrace{(\text{new}\, a)(!b\langle a\rangle.?a(y).P\langle x\rangle)}_{P'},\\
Q(x) &\overset{\text{def}}{=} \underbrace{?b(y).(\text{new}\, b)(Q\langle y\rangle)}_{Q'}.
\end{aligned}$$

Then

$$\begin{aligned}
\text{fn}(P') &= \{b, x\}, & \text{bn}(P') &= \{a, y\},\\
\text{fn}(Q') &= \{b\}, & \text{bn}(Q') &= \{y, b\},\\
\text{fn}(P'|P') &= \text{fn}(P'), & \text{bn}(P'|P') &= \text{bn}(P').
\end{aligned}$$

Channel continuations of the form $?a(x).P$ are capable of receiving a message m. The effect of this is that the variable x *bound* to the action $?a(x)$ gets "replaced" by m in P. We formalize this in the following definition.

Definition (Substitution)**.** We define a substitution function $\cdot\{\cdot \mapsto \cdot\} \colon \mathcal{P} \times \mathcal{V} \times (\mathcal{V} \cup \mathcal{N}) \to \mathcal{P}$ as $\cdot\{\cdot \mapsto \cdot\}$

(i) $\mathbf{0}\{x \mapsto y\} = \mathbf{0}$,

(ii) $(!a\langle b\rangle.P)\{x \mapsto y\} = !c\langle d\rangle.(P\{x \mapsto y\})$ where

$$c = \begin{cases} y & \text{if } x = a,\\ a & \text{otherwise,}\end{cases} \qquad d = \begin{cases} y & \text{if } x = b,\\ b & \text{otherwise,}\end{cases}$$

(iii) $$(?a(z).P)\{x \mapsto y\} = \begin{cases} ?a(z).P & \text{if } x = z,\\ (?a(w).(P\{z \mapsto w\}))\{x \mapsto y\} & \text{if } y = z,\ w \notin \text{fn}(P),\\ ?y(z).(P\{x \mapsto y\}) & \text{if } x = a,\\ ?a(z).(P\{x \mapsto y\}) & \text{otherwise,}\end{cases}$$

(iv) $(\sum_{i\in I} \alpha_i.P_i)\{x \mapsto y\} = \sum_{i\in I}(\alpha_i.P_i)\{x \mapsto y\}$,

(v) $(P|Q)\{x \mapsto y\} = P\{x \mapsto y\}|Q\{x \mapsto y\}$,

(vi) $A\langle b\rangle\{x \mapsto y\} = A\langle y\rangle$ if $b = x$ and $A\langle b\rangle$ otherwise,

(vii) $$((\text{new}\, z@r)P)\{x \mapsto y\} = \begin{cases} (\text{new}\, z@r)P & \text{if } x = z,\\ ((\text{new}\, w@r).P\{z \mapsto w\})\{x \mapsto y\} & \text{if } y = z, \text{ where } w \notin \text{fn}(P),\\ (\text{new}\, z@r)P\{x \mapsto y\} & \text{otherwise.}\end{cases}$$

This definition naturally extends to the case when the channel names a, b and variables z are replaced by vectors of channel names and variables – we can use the shorthand

$$P\{\widetilde{x} \mapsto \widetilde{\psi}\} \text{ for } P\{x_1 \mapsto \psi_1\}\{x_2 \mapsto \psi_2\}\cdots\{x_n \mapsto \psi_n\}$$

where $\widetilde{x} = (x_1, x_2, \ldots, x_n) \in \mathcal{V}^n$ and $\widetilde{\psi} = (\psi_1, \psi_2, \ldots, \psi_n) \in (\mathcal{V} \cup \mathcal{N})^n$ for some $n \in \mathcal{N}$.

Example 4. By the above definition, we have for example

$$\begin{aligned}
(?a(x).A\langle x\rangle + !a\langle x\rangle.B\langle x\rangle)\{x \mapsto c\} &= ?a(x).A\langle x\rangle + !a\langle c\rangle.B\langle c\rangle,\\
((\text{new}\, b@r)A\langle b, c\rangle)\{c \mapsto d\} &= (\text{new}\, b@r)A\langle b, d\rangle,\\
((\text{new}\, b@r)!a\langle b\rangle)\{a \mapsto b\} &= (\text{new}\, c@r)!b\langle c\rangle.
\end{aligned}$$

In the above definition, in cases (iii) and (vi), the new name w can be arbitrary. More generally, the actual bound name used with the new operator and the variable within an input action should not matter, as long as it stays private for the restriction.

$\cdot \equiv_\alpha \cdot$ **Definition** (α congruence)**.** We say that two processes P and Q are *alpha congruent*, write $P \equiv_\alpha Q$, if they differ only in the choice of their bound names.

Example 5. By the above definition, we have

$$\begin{aligned}(\mathrm{new}\, a@r)!b\langle a\rangle &\equiv_\alpha (\mathrm{new}\, c@r)!b\langle c\rangle,\\ (\mathrm{new}\, a@r)!b\langle a\rangle &\equiv_\alpha (\mathrm{new}\, a@r)!b\langle a\rangle,\\ (\mathrm{new}\, x@r)!a\langle x\rangle|(\mathrm{new}\, x@r)!a\langle x\rangle &\equiv_\alpha (\mathrm{new}\, y@r)!a\langle y\rangle|(\mathrm{new}\, z@r)!a\langle z\rangle,\\ ?a(x).!x &\equiv_\alpha ?a(y).!y.\end{aligned}$$

3.3 Semantics

We are now ready to describe the behaviour of π processes. Intuitively, two processes present within a parallel composition are able to communicate if each can execute a complementary (input to output and vice versa) action on the same channel. This gets more complicated if the outputting process within a restriction is sending the new name – the receiver has to be put under the same restriction, otherwise the name would no longer stay unique to the restriction.

We need to distinguish when an output action is sending a name that is bound by an enclosing restriction.

$(\varphi)!a\langle\widetilde{\psi}\rangle$ **Definition.** A *bound output action* is of the form $(\varphi)!a\langle\widetilde{\psi}\rangle$ where $\varphi \subseteq \mathcal{N}$, $\varphi \subseteq \widetilde{\psi}$. If φ is the empty set, this becomes a normal output action $!a\langle\widetilde{\psi}\rangle$.

$?a\langle\widetilde{\psi}\rangle$ A *free input action* is of the form $?a\langle\widetilde{\psi}\rangle$ where $\widetilde{\psi} \in (\mathcal{V} \cup \mathcal{N})^n$ for some $n \in \mathbb{N}$.

$\mathcal{A}_+$ Let $\mathcal{A}_+$ be the set of actions extended with bound output actions and free input actions.

$\mathrm{fn}(\alpha)$ Extend the function fn to actions in the obvious way, so that $\mathrm{fn}(!a\langle\widetilde{\psi}\rangle) = \mathrm{fn}(?a\langle\widetilde{\psi}\rangle) = \{a\} \cup \widetilde{\psi}$, $\mathrm{fn}(?a(\widetilde{x})) = \{a\}$, $\mathrm{fn}((\varphi)!a\langle\widetilde{\psi}\rangle) = \{a\} \cup \widetilde{\psi} \setminus \varphi$.

$\mathrm{on}(\cdot)$ Define the *open names* of an action, $\mathrm{on}\colon \mathcal{A}_+ \to 2^{\mathcal{V}\cup\mathcal{N}}$ as $\mathrm{on}((\varphi)!a\langle\widetilde{\psi}\rangle) = \varphi$ and $\emptyset$ otherwise.

We now give the semantics of $\mathcal{S}\pi$ in form of a *multi-transition system* – a directed graph allowing multiple edges and loops. The states will correspond to $\mathcal{S}\pi$ processes and the transitions to the possible behaviour. The need for this to be a *multi*-transition system arises from the fact that different behaviour can lead to the same evolution of the system.

$\cdot \overset{\cdot}{\longrightarrow} \cdot$ **Definition** (transition semantics)**.** The *transition relation* is a multi-relation $\cdot \overset{\cdot}{\longrightarrow} \cdot \Subset \mathcal{P}\times\mathcal{A}_\tau\times\mathcal{P}$, restriction of the multi-relation $\cdot \overset{\cdot}{\longrightarrow} \cdot \Subset \mathcal{P}\times\mathcal{A}_+\times\mathcal{P}$ inductively defined by the rules in the Figure 3.1.

The rule Act expresses that an output or silent action continuations evolve into the process prefixed by that action. For inputs, this also depends on the actual message received – the rule In expresses all the possibilities. These two rules are expressing the basic notions of *capability* and are only auxiliary to defining the transition relation (the transition pairs they define obviously belong to the difference of the pre-transition relation and the transition relation).

The rule Ide allows for identifiers instances to be treated in the same way as the processes they are defining.

Together, these three rules form base cases for the further rules. The rule Sum gives a summation the capability of all the possible actions of its individual continuations (one at a time) and the rule Par allows components of parallel composition to evolve individually. The rule Com synchronizes the evolution of two processes capable of complementary input and output action within a parallel composition. This turns capabilities into transitions in the transition relation.

The rule Res treats the case when a process inside a restriction is capable of an action not involving the private channel – in that case the same capability is passed on the restriction.

The rule Open handles the case when the private channel name can be sent out, by lifting the restriction and storing the private name in the bound output action of the resulting capability. Finally, the rule Close synchronizes two processes as the rule Par with additionally enclosing the resulting parallel composition in a restriction on the channel names stored in the involved bound output action.

$$\begin{array}{ll}
\textsc{Act}: & \alpha.P \xrightarrow{\alpha} P,\ \alpha \text{ not input}, \\[2ex]
\textsc{In}: & ?a(\widetilde{x}).P \xrightarrow{?a\langle\widetilde{\psi}\rangle} P\{x \mapsto \widetilde{\psi}\} \\[2ex]
\textsc{Ide}: & \dfrac{P\{\widetilde{x} \mapsto \widetilde{\psi}\} \xrightarrow{\alpha} P'}{A\langle\widetilde{\psi}\rangle \xrightarrow{\alpha} P'},\ A(\widetilde{x}) \overset{\text{def}}{=} P \\[2ex]
\textsc{Par}: & \dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q},\ \text{fn}(Q) \cap \text{on}(\alpha) = \emptyset \\[2ex]
\textsc{Sum}: & \dfrac{\alpha_j.P_j \xrightarrow{\alpha_j} P',\ j \in I}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{\alpha} P'} \\[2ex]
\textsc{Res}: & \dfrac{P \xrightarrow{\alpha} P'}{(\text{new}\, b@r)P \xrightarrow{\alpha} (\text{new}\, b@r)P'},\ b \notin \text{fn}(\alpha),\ b \notin \text{on}(\alpha) \\[2ex]
\textsc{Open}: & \dfrac{P \xrightarrow{(\varphi)!a\langle\widetilde{\psi}\rangle} P'}{(\text{new}\, b@r)P \xrightarrow{(\varphi\cup\{b@r\})!a\langle\widetilde{\psi}\rangle} P'},\ b \in \widetilde{\psi},\ b \notin \varphi \\[2ex]
\textsc{Close}: & \dfrac{P \xrightarrow{(\varphi)!a\langle\widetilde{\psi}\rangle} P' \qquad Q \xrightarrow{?a\langle\widetilde{\psi}\rangle} Q'}{P|Q \xrightarrow{\tau@r_a} (\text{new}\,\varphi)(P'|Q')} \\[2ex]
\textsc{Com}: & \dfrac{P \xrightarrow{!a\langle\widetilde{\psi}\rangle} P' \qquad Q \xrightarrow{?a\langle\widetilde{\psi}\rangle} Q'}{P|Q \xrightarrow{\tau@r_a} P'|Q'}
\end{array}$$

Figure 3.1: The transition rules for the stochastic π calculus. The rules Par, Close and Com also have alternatives with the operands of the | operator swapped.

Example 6. Figure 3.2 gives a sample derivation using these rules. Figure 3.3 gives an example of all the possible capabilities and transitions of a process.

When we eventually get a continuous-time Markov chain (CTMC) from the transition system and simulate it, in each state we need to consider all the possible transitions.

Definition. For a process $P \in \mathcal{P}$, define the multiset of *possible transitions* $\mathit{Trans}(P) \in \mathcal{M}(\mathcal{P} \times \mathcal{A}_\tau \times \mathcal{P})$ to be $\mathit{Trans}(P)$

$$\mathit{Trans}(P) = \{\!|(P, \tau@r, P') : P \xrightarrow{\tau@r} P'|\!\}.$$

We will write the elements of $\mathit{Trans}(P)$ as $P \xrightarrow{\tau@r} P'$ and usually replace the $\tau@r$ by only r.

We will also write $P \xrightarrow{\alpha_1\alpha_2\cdots\alpha_n} P'$, $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathcal{A}$ to abbreviate the statement that there exist processes $P_1, \ldots, P_{n+1}$ such that $P_1 = P$, $P_{n+1} = P'$ and $P_i \xrightarrow{\alpha_i} P_{i+1}$ for $i = 1, \ldots, n$.

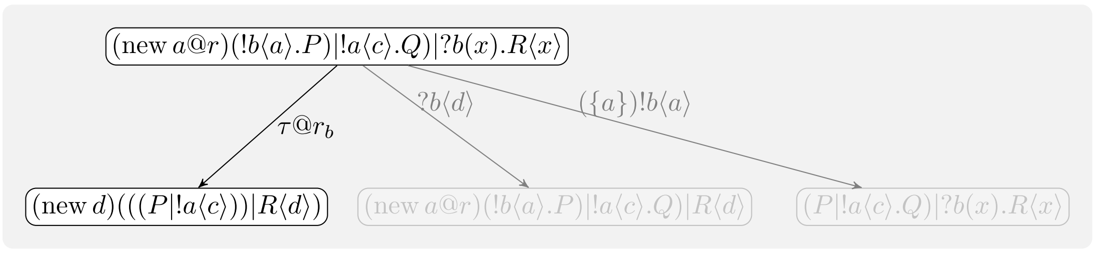
Define the set of *derivatives* of P, $\mathit{Ds}(P)$ to be $\mathit{Ds}(P)$

$$\mathit{Ds}(P) = \{P' : P \xrightarrow{\tau@r_1\tau@r_2\cdots\tau@r_n} P' \in \mathit{Trans}(P)\}.$$

Finally, define the *transition diagram* of P to be the multi-graph with vertices members of $\mathit{Ds}(P)$ and oriented edges (with labels) members of the transition relation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{!b\langle d\rangle.P \xrightarrow{!b\langle d\rangle} P}\,\textsc{Act}}{(\text{new}\, d@r)(!b\langle d\rangle.P) \xrightarrow{(d)!b\langle d\rangle} P}\,\textsc{Open}}{(\text{new}\, d@r)(!b\langle d\rangle.P|!a\langle c\rangle.Q) \xrightarrow{(d)!b\langle d\rangle} P|!b\langle c\rangle}\,\textsc{Par} \qquad \dfrac{}{?b(x).R\langle x\rangle \xrightarrow{?b\langle d\rangle} R\langle d\rangle}\,\textsc{In}}{((\text{new}\, a@r)(!b\langle a\rangle.P)|!a\langle c\rangle.Q)|?b(x).R\langle x\rangle \xrightarrow{\tau@r_b} (\text{new}\, d)(((P|!a\langle c\rangle))|R\langle d\rangle)}\,\textsc{Close}$$

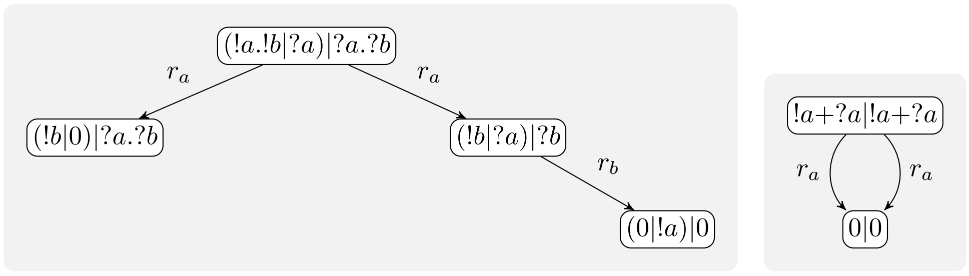
Figure 3.2: Example of a derivation for a member of the transition relation. The transition is built from two complementing capabilities, which are built on the structure of the corresponding processes, with the base cases expressed by the rules Act and In.



$(\text{new}\, a@r)(!b\langle a\rangle.P)|!a\langle c\rangle.Q)|?b(x).R\langle x\rangle$
$\tau@r_b$
$?b\langle d\rangle$
$(\{a\})!b\langle a\rangle$
$(\text{new}\, d)(((P|!a\langle c\rangle))|R\langle d\rangle)$
$(\text{new}\, a@r)(!b\langle a\rangle.P)|!a\langle c\rangle.Q)|R\langle d\rangle$
$(P|!a\langle c\rangle.Q)|?b(x).R\langle x\rangle$

Figure 3.3: All the possible (pre-)transitions of the process $(\text{new}\, a@r)(!b\langle a\rangle.P)|!a\langle c\rangle.Q)|?b(x).R\langle x\rangle$. The capabilities (i.e. the pre-transitions that are not transitions) are faded.

Example 7. See Figure 3.4 for an example of a transition diagram and an example of why the set of possible transitions has to be a multiset (and so also why the transition relation has to be a multi-relation).



(a) An example of a transition diagram.

(b) An example demonstrating the need for $\mathit{Trans}(P)$ to be a multiset.

Figure 3.4: Transition diagrams. Figure (a) shows the complete transition diagram of a process and figure (b) demonstrates why the transition diagrams can be multi-graphs.

The transitions diagram of P can be now seen as a description of a CTMC. We can associate each node with a state of the chain and each edge with a transition between states, with the rate of the exponential being the edge's label. This gives a CTMC which can be simulated by one of the algorithms mentioned in the previous chapter. However, there is a drawback to this approach. In biochemical applications of $\mathcal{S}\pi$, we usually have large numbers of certain processes (say those modelling molecules). If this is the case, we can see that the number of states, and more importantly of transitions, will be large. If we consider a process P that allows a silent transition and a system with n copies of process P put in parallel (with the bracketing chosen arbitrarily), then we can see that there are n possible silent transitions to a system with $(n-1)$

copies of P, each bracketed differently. This is not desired for two reasons. First, to list all the transitions of the system (as is needed in each iteration of Algorithm 1), we would need to traverse the expression of length n. Secondly, if P processes correspond to molecules, we are usually not distinguishing between two different copies of P (as for the Gillespie algorithm to apply, we assume that the molecules are uniformly distributed across the volume) and would rather like to consider each bracketing of the system to be the same. This is a target of some criticism of stochastic π calculus, [12].

The following section defines a congruence relation that will, apart from other things, abstract from the order and bracketing of parallel compositions and allow an efficient enumeration of all the possible transitions of a system, not dependent on the numbers of individual parallel components.

3.4 Structural congruence

As a next step, we abstract further from the syntax of stochastic π calculus by considering processes modulo an equivalence relation – the *structural congruence*. The properties of this relation allow us to describe the semantics in a more succinct form and eventually enable more efficient simulation.

Definition (structural congruence)**.** We say that two processes are *structuraly congruent* if they belong to the relation $\cdot \equiv \cdot \subseteq \mathcal{P} \times \mathcal{P}$ inductively defined as the least equivalence relation preserved by the process constructs and satisfying $\cdot \equiv \cdot$

(i) $P|Q \equiv Q|P$,

(ii) $(P|Q)|R \equiv P|(Q|R)$,

(iii) $P|\mathbf{0} \equiv P$,

(iv) $P \equiv Q$ if $P \equiv_\alpha Q$,

(v) $(\text{new}\, a@r)(\text{new}\, b@s)P \equiv (\text{new}\, b@s)(\text{new}\, a@r)P$,

(vi) $(\text{new}\, a@r)\mathbf{0} \equiv \mathbf{0}$,

(vii) $((\text{new}\, a@r)P)|Q \equiv (\text{new}\, a@r)(P|Q)$ if $a \notin \text{fn}(Q)$,

(viii) $A\langle\widetilde{\psi}\rangle \equiv P\{\widetilde{x} \mapsto \widetilde{\psi}\}$ if $A(\widetilde{x}) \overset{\text{def}}{=} P \in E$.

The rule (v) allows us to extend the notation of the new operator to sets. From now on, we will write $(\text{new}\, \psi)P$, where $\psi = \{a_1@r_1, \ldots, a_n@r_2\} \subseteq \mathcal{N}$, to represent the congruence class of $(\text{new}\, a_1@r_1) \cdots (\text{new}\, a_n@r_n)P$. Note that it is sufficient for ψ to be a set and not a multiset – $(\text{new}\, a@r)(\text{new}\, a@r)P \equiv (\text{new}\, a@r)P$ since $a \notin \text{fn}((\text{new}\, a@r)P)$.

Similarly, the rules (i) and (ii) allow us to extend the notation of the parallel composition operator to multisets – we will write

$$\{\!|n_1 \times P_1, n_2 \times P_2, \cdots, n_m \times P_m|\!\}$$

to represent the congruence class of

$$\underbrace{P_1|P_1|\cdots|P_1}_{n_1}|\cdots|\underbrace{P_m|\cdots|P_m}_{n_m}$$

We can think of this as bringing the syntactical representation closer to the models where we assume that the individual species (such as molecules) are uniformly distributed in the enclosing compartment ("represented" by the multiset).

From now on we will use the multiset representation and the $(\text{new}\, \psi)$ notation interchangeably for processes. Formally, whenever such representation appears where a process should, we choose any process belonging to the congruence class of the representation. For example, when say $\{\!|2 \times P, Q|\!\}$ is a process, we mean any process of the congruence class of $\{\!|2 \times P, Q|\!\}$, that is for example $(P|P)|Q$.

The following theorem shows that the structural congruence preserves the semantics of processes. We need this to justify that we can use structurally congruent processes interchangeably in the transition systems and hence in the resulting CTMCs.

Theorem 3.1. For two processes $P, Q \in \mathcal{P}$, if $P \equiv Q$, then there is a multiset bijection $\phi\colon Ds(P) \to Ds(Q)$ such that if $P \xrightarrow{\alpha_1\alpha_2\cdots\alpha_n} P'$ then $Q \xrightarrow{\alpha_1\alpha_2\cdots\alpha_n} Q'$ and $P' \equiv Q'$ for $Q' = \phi(P')$.

Proof. We can prove a stronger version of this statement extended to derivatives of P, Q under the full pre-transition relation. The proof would proceed by induction on the structure of $\equiv$. ∎

3.5 Simulation

The above theorem let us consider processes modulo the structural congruence when reasoning about the simulation. The next natural step is to choose a representative of classes of structural congruence. This consists of describing the "form" of this representative and then showing that such form belongs to each congruence class.

Definition (standard form)**.** A *standard form* is an equivalence class represented by

$$(\text{new}\,\psi)\{\!|n_1 \times P_1, \cdots, n_m \times P_m|\!\}$$

where ψ is a finite set of names, $n_i \in \mathbb{N}$ and $P_i \in \mathcal{P}$, P_i is a summation.

Note that our definition of standard form differs from that of [27], where it is a *process.* We abstract from this and take it to be a multiset representation.

Proposition 3.2. Every process is structurally congruent to a standard form.

Proof. Take a process $P \in \mathcal{P}$. First assume that the syntax tree of P has identifier instances only under summations. Then by structural induction on the formation of P:

(i) $P = \mathbf{0}$ is congruent to $(\text{new}\,\emptyset)\{\!||\!\}$.

(ii) $P = \sum_{i\in I} \alpha_i.P_i$ is congruent to $(\text{new}\,\emptyset)\{\!|P|\!\}$.

(iii) $P = P_1|P_2$ is congruent to $(\text{new}\,\psi_1 \cup \psi_2)M_1 \uplus M_2$ where $(\text{new}\,\psi_1)M_1$ and $(\text{new}\,\psi_2)M_2$ are standard forms of P_1 and P_2 respectively such that $\psi_1 \cap \psi_2 = \emptyset$ (their existence i iss guaranteed by the inductive hypothesis and the fact that we can use alpha congruence to get sets of names with the desired property).

(iv) $P = (\text{new}\,a@r_a)P$ belongs to $(\text{new}\{a@r_a\} \cup \psi)M$ where $(\text{new}\,\psi)M$ is a standard form of P such that $a \notin \psi$.

Let $A_i(\widetilde{x}) \overset{\text{def}}{=} P_i$, $i = 1, \ldots, n$ be all the defining equations in the given valid environment. Because the environment is valid, none of the identifiers immediately produces itself and therefore the identifiers can be ordered as $A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ such that A_{i_j} does not immediately produce A_{i_k} for $k > j$ and $j = 1, \ldots, n$. Then P_{i_1} contains identifier instances only up to under summations and so $A_{i_1}\langle\widetilde{\psi}\rangle$ is structurally congruent to a standard form for any $\widetilde{\psi} \in (\mathcal{V} \cup \mathcal{N})^m$. Now $A_{i_2}\langle\widetilde{\psi}\rangle$ is structurally congruent to a standard form, since P_{i_2} will contain at most A_{i_1} not under a summation. We can proceed with all the identifiers upto A_{i_n} and therefore every process is structurally congruent to a standard form. ∎

The main aim of this section is to give a complete enumeration of all the possible transitions of a process. For processes that are summations, this enumeration is straightforward – the only possible rule that can be applied is Sum.

In case of a tree of nested parallel compositions, it is easy to see that each process can potentially communicate with any other process in the tree (except for itself), through multiple applications of the rule Par. This also means that if there are several copies of a process in the tree, they can all communicate with the same processes. For example, take the process $(Q|(P|P))|(Q|P)$ and assume that processes P, Q can communicate (e.g. P is capable of an output action on channel a and Q of an input action on a). We can label each occurrence of P,Q and the $|$ operator, say as $(Q^1|^2(P^3|^4P^5))|^6(Q^7|^8P^9)$. Then each occurrence of P can communicate with each occurrence of Q, "through" one of the $|$ operators. See Figure 3.5 for an illustration.

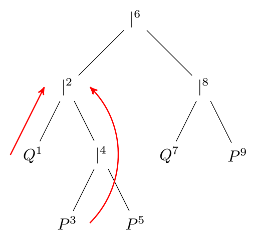


Figure 3.5: The occurence P^3 can communicate with Q^1 with PAR applied at the level of $|^4$ and with COM applied at the level of $|^2$. Similarly P^5 can communicate with Q^1 through I^2, P^3 with Q^7 with PAR applied at the levels of $|^8$ and $|^2$ and COM at the level of $|^6$, etc. In total, there are $3 \cdot 2$ ways in which a P can communicate with a Q.

Therefore there are exactly $3 \cdot 2$ possible ways P and Q can communicate. In general, if there are n occurrences of P and m occurrences of Q in the tree of nested parallel compositions (corresponding to a process R), there are $n \cdot m$ ways in which P and Q can communicate and so *Trans*(R) will contain $n \cdot m$ copies of the transition $R \xrightarrow{\tau @ r_a} R'$.

We can derive a similar result for the case when P can communicate with another P, with the difference that the total number of transitions will be $n \cdot (n-1)$ since a process cannot communicate with itself.

Finally, n copies of a process capable of a silent action will result in n silent transitions.

The following theorem summarizes the above mentioned enumeration, based on the standard form of a process.

Theorem 3.3. If P has a standard form $(\text{new}\,\psi)P'$, where $P' \equiv \{\!|n_1 \times P_1, \cdots, n_m \times P_m|\!\}$, then

$$\begin{aligned}
\mathit{Trans}(P) = \{\!| r(i,j) \times P \xrightarrow{\tau @ r_a} (\text{new}\,\psi)(P' \setminus P_i \setminus P_j \uplus Q_i \uplus Q_j\{\widetilde{x} \mapsto \widetilde{\psi}\})\\
: P_i \xrightarrow{!a\langle\widetilde{\psi}\rangle} Q_i,\ P_j \xrightarrow{?a(\widetilde{x})} Q_j |\!\} \uplus\\
\{\!| n_i \times P \xrightarrow{\tau @ r} (\text{new}\,\psi)(P' \setminus P_i \uplus Q_i) : P_i \xrightarrow{r} Q_i |\!\}
\end{aligned}$$

where

$$r(i,j) = \begin{cases} n_i \times n_j & \text{if } i \neq j,\\ n_i \times (n_i - 1) & \text{otherwise.}\end{cases}$$

Proof. We choose a process from the congruence class of P and show that all the possible transitions defined in the semantics agree with the given enumeration (as is argued in the paragraph above). The result then follows from the Theorem 3.1. ∎

Note that we use the capability $P_i \xrightarrow{!a\langle\widetilde{\psi}\rangle} Q_i$ only for convenience and could have said that P_i is a summation where one of the summands of $!a\langle\widetilde{\psi}\rangle.Q_i$.

Example 8. Take an environment containing the defining equations

$$\begin{aligned}
&P \overset{\text{def}}{=} !a.P_1 + \tau @ r_1.P_2,\\
&Q \overset{\text{def}}{=} ?a.Q_1,\\
&R \overset{\text{def}}{=} ?b.R_1 + !b.R_2,\\
&S \overset{\text{def}}{=} 3 \times P | 4 \times Q | 5 \times R.
\end{aligned}$$

Then we have *Trans*(S) containing precisely the following transitions

$$\begin{aligned}
12 \times (S \xrightarrow{\tau @ r_a} 2 \times P | 3 \times Q | 5 \times R | P_1 | Q_1),\\
20 \times (S \xrightarrow{\tau @ r_n} 3 \times P | 4 \times Q | 3 \times R | R_1 | R_2),\\
3 \times (S \xrightarrow{\tau @ r_1} 2 \times P | 4 \times Q | 5 \times R | P_2).
\end{aligned}$$

The Theorem 3.3 allows us to aggregate structurally congruent processes and efficiently compute all the possible transitions. This is projected into an aggregation of the state space of the resulting CTMC and also to an efficient enumeration of the transitions from each state. This leads to an efficient simulation algorithm, see Algorithm 4.

Also, we can see the Theorem 3.3 as justifying the fact that $\mathcal{S}\pi$ naturally gives the mass-action kinetics – the rate of communication between two processes is equal to the product of their populations in the system. This is not completely true if the two processes are of the same kind, when the rate is $n \cdot (n-1)$ if the population is n. However, this difference can be neglected if n is large, which is the case when mass-action kinetics applies.

Algorithm 4 The Gillespie algorithm for $\mathcal{S}\pi$.

```
1: Start with a top-level process $S$
2: repeat
3:   get the standard form of $S$ to be $(\text{new}\,\varphi)\,\underbrace{\{\!|n_1 \times P_1, \ldots, n_m \times P_m|\!\}}_{S'}$
4:   for all $i$, $j$, $a \in \varphi \cup \text{fn}(S)$ do {collect all transitions $T = \{\!|S \xrightarrow{\tau@r} U|\!\}$}
5:     if $P_i \xrightarrow{!a\langle\tilde{\psi}\rangle} Q_i$ and $P_j \xrightarrow{?a\langle\tilde{\psi}\rangle} Q_j$ then
6:       Let $(\text{new}\,\varphi_i)M_i$ be a standard form of $Q_i$ and $(\text{new}\,\varphi_j)M_j$ be a standard form of $Q_j$,
         with $\varphi_1 \cap \varphi_1 = \varphi_1 \cap \varphi = \varphi_2 \cap \varphi = \emptyset$
7:       insert $r(n_i, n_j)$ copies of $S \xrightarrow{\tau@r_a} (\text{new}\,\varphi \cup \varphi_i \cup \varphi_j)(S' \setminus P_i \setminus P_i \uplus M_i \uplus M_j)$ into $T$, where
         $r(n_i, n_j) = n_i \cdot n_j$ if $i \neq j$ and $n_i \cdot (n_i - 1)$ otherwise
8:     end if
9:     if $P_i \xrightarrow{\tau@r} Q_i$ then
10:      Let $(\text{new}\,\varphi_i)M_i$ be a standard form of $Q_i$, $\varphi_i \cap \varphi = \emptyset$
11:      insert $n_i$ copies of $S \xrightarrow{\tau@r} (\text{new}\,\varphi \cup \varphi_i)(S \setminus P_i \uplus M_i)$ into $T$
12:    end if
13:  end for
14:  let $r_{\text{total}} = \sum_{S \xrightarrow{\tau@r} U} r$
15:  randomly select a transition $S \xrightarrow{\tau@r} U$ with probability $r/r_{\text{total}}$
16:  generate $\delta t$ from $\text{Exp}(r_{\text{total}})$
17:  set $t = t + \delta t$, set $S = U$
18: until until $t = t_{\text{stop}}$
```

The efficiency improvement of the state space aggregation is obvious. Let the standard form of the top-level process in each iteration contain n summations out of which there are m different ones, with k the maximal size of an index set. Then each iteration has running time $O(m^2k^2)$, as opposed to $O(n^2k^2)$ in case the summations are not aggregated. This is a significant performance improvement, as the models often contain hundreds of identical processes.

3.6 Prime processes

In this section we define a different representative of the equivalence classes of structural congruence, that will be useful when defining the continuous semantics of $\mathcal{S}\pi$. The basic blocks will be *prime processes* – processes representing the defined species or complexes of them which cannot be divided.

Definition. A process of $\mathcal{S}\pi$, $P \in \mathcal{P}$ is a *prime process* if $P \equiv Q|R$ for some $Q, R \in \mathcal{P}$ implies that $Q \equiv \mathbf{0}$ or $R \equiv \mathbf{0}$.

$\widehat{\mathcal{P}}$ Denote the set of all prime processes by $\widehat{\mathcal{P}}$.

Definition. Given a process $P \in \mathcal{P}$, the *prime decomposition* of P is $P \equiv \{\!|n_1 \times P_1, \ldots, n_m \times P_m|\!\}$ where all P_i are prime processes, not structurally congruent.

The following Proposition, like the Proposition 3.2, shows that the prime decomposition is a well defined representative of congruence classes.

Proposition 3.4. Every process P of $\mathcal{S}\pi$ has a unique (up to structural congruence), finite, prime decomposition.

Proof. In [25], author suggests assigning normal forms to processes using a normalising and confluent term rewriting system respecting $\equiv$, and taking the prime decomposition as the multiset of parallel components of the normal form. ■

Originally, the only restriction on the definition of identifiers was that they cannot immediately produce themselves. We add another restriction, that is identifiers can be only defined as prime processes. This will make certain technical details in the following easier.

3.7 Summary

We introduced and formally defined stochastic π calculus ($\mathcal{S}\pi$). We gave a formal definition of its semantics and shown why it is not directly suitable for efficient simulation. We proceeded with defining the structural congruence and shown how it can be used for state aggregation in the underlying CTMC and we gave an efficient simulation algorithm for $\mathcal{S}\pi$ based on the Gillespie algorithm.

4 Continuous semantics of stochastic π calculus

The main aim of this chapter is to provide an alternative semantics of stochastic π calculus. We will refer to the original semantics defined in the previous chapter as to *discrete* and *stochastic*. We concluded the last chapter by defining the prime decomposition of processes. This enables to express each process as a vector (possibly infinitely dimensional) of populations of prime processes. These populations are always integers (hence the term discrete for the original semantics). Each process is then capable of evolving into a different process, by performing a silent transition (which either results from a communication between two subprocesses or a capability of a subprocess to perform a silent action). Each such transition is interpreted as having an exponential delay with its given rate (hence the term *stochastic*) and so the behaviour of the system can be described by a continuous time Markov chain (CTMC).

Discrete stochastic simulation is often used as an alternative to the more traditional approach of modelling with a system of ODEs (which describe the model in a *continuous* and *deterministic* way), as we already mentioned when describing the Gillespie algorithm. Therefore, one can still be interested in the original approach to the problem. Inherently, the stochastic π calculus language (and other stochastic process algebras, such as PEPA) does not necessitate the discrete stochastic semantics. Instead, it can be thought of as a intermediate description of the model on which different analyses can be performed. A recent tendency is to provide an alternative semantics in form of a system of ODEs obtained from the syntactical description of the model, that somehow corresponds to the time evolution of the system. In [7], this semantics is provided for a subset of stochastic π calculus called *Chemical Ground Form* (CGF), via translation using chemical reactions. We provide an equivalent formulation extended to the full $\mathcal{S}\pi$, using similar style to [20], where a continuous (or *fluid*) semantics is defined for the PEPA process algebra.

We will show that this semantics gives a finite set of ODEs if the system is of CGF.

Then in the next chapter, we will also highlight several (non-CGF) models where the obtained system of ODEs is infinitely large and therefore not viable for a numerical solution. Motivated by this, we try to formulate conditions on the models which ensure that the set of ODEs will be finite.

Finally, we try to compare the results from both of the semantics and experimentally verify properties similar to those in [14].

We first define CGF, a subset of $\mathcal{S}\pi$, and also of CCS, which is equivalent to basic chemistry[10].

Definition. *Chemical Ground Form* (CGF) is a subset of stochastic π calculus, with processes $\mathcal{P}_{\text{CGF}} \subseteq \mathcal{P}$ a minimal set such that *CGF*

(i) $\mathbf{0} \in \mathcal{P}_{\text{CGF}}$, $\mathbf{0}$

$\sum_{i\in I}\alpha_i.P_i$ (ii) $\sum_{i\in I}\alpha_i.P_i \in \mathcal{P}_{\mathrm{CGF}}$, where I is a finite index set and α_i a channel action without arguments or a silent action and $P_i \in \mathcal{P}_{\mathrm{CGF}}$ a process for all $i \in I$,

$P|Q$ (iii) $P, Q \in \mathcal{P}$, $P|Q \in \mathcal{P}_{\mathrm{CGF}}$

$A\langle\widetilde{\psi}\rangle$ (iv) $A\langle\widetilde{\psi}\rangle \in \mathcal{P}_{\mathrm{CGF}}$, $\widetilde{\psi} \in (\mathcal{N}\cup\mathcal{V})^n$ for some $n \in \mathbb{N}$.

We can restrict the definition of an environment for stochastic π calculus to an environment for CGF and also define the restricted operational semantics and structural congruence.

Prime processes will correspond to all the different "species" arising during the evolution of the model. For the whole $\mathcal{S}\pi$ (and also CGF), the set of prime processes $\widehat{\mathcal{P}}$ is infinite (consider simple continuations of the form $!a$ for a name $a \in \mathcal{N}$). However, when modelling with a fixed system, we are only interested in the prime processes that are reachable from the initial top-level process.

$\widehat{\mathcal{P}}(S,E)$ **Definition.** Prime processes of a system (S,E) is the set $\widehat{\mathcal{P}}(S,E)$ of all prime processes P such that $S \longrightarrow^* P|Q$ for some Q.

In case of CGF, there is an efficient algorithm enumerating the possible prime processes of a system, which depends explicitly only on the syntactical structure of the environment. Let $\widehat{\mathcal{P}}_{\mathrm{CGF}}$ denote all the prime processes of CGF.

$\widehat{\mathcal{P}}^*_{CGF}(S,E)$ **Definition.** Given a CGF system (S,E), define the *(CGF) prime approximation* of (S,E), $\widehat{\mathcal{P}}^*_{\mathrm{CGF}}(S,E) \subseteq \mathcal{P}_{\mathrm{CGF}}$ as

$$\widehat{\mathcal{P}}^*_{\mathrm{CGF}}(S,E) = \bigcup_{i=1}^{m} \widehat{\mathcal{P}}_E(P_i)$$

where $\{\!|n_1 \times P_1, \ldots, n_m \times P_m|\!\}$ is a prime decomposition of S and $\widehat{\mathcal{P}}_E \colon \mathcal{P}_{CCS} \to 2^{\mathcal{P}_{\mathrm{CGF}}}$ is inductively defined as

(i) $\widehat{\mathcal{P}}_E(\mathbf{0}) = \emptyset$,

(ii) $\widehat{\mathcal{P}}_E(\sum_{i\in I}\alpha_i.P_i) = \bigcup_{i\in I}\widehat{\mathcal{P}}_E(P_i) \cup \{P_i\}$,

(iii) $\widehat{\mathcal{P}}_E(A\langle\widetilde{\psi}\rangle) = \widehat{\mathcal{P}}_E(P\{\widetilde{x} \mapsto \widetilde{\psi}\}) \cup \{A\langle\widetilde{\psi}\rangle : P \text{ is prime}\}$, $A(\widetilde{x}) \overset{\mathrm{def}}{=} P \in E$,

(iv) $\widehat{\mathcal{P}}_E(P|Q) = \{R : R = P, Q \text{ and } R \text{ is not a parallel composition}\} \cup \widehat{\mathcal{P}}_E(P) \cup \widehat{\mathcal{P}}_E(Q)$.

The following proposition justifies the choice of the prime approximation. Within a fixed environment in CGF, it is possible to have only a finitely many different types of prime processes. This results from the fact that the only way to "create" a new process in a transition is by parallel composition, which by definition cannot give rise to a new prime process other than those defined in the environment.

Proposition 4.1. For all CGF systems (S,E), $\widehat{\mathcal{P}}(S,E) \subseteq \widehat{\mathcal{P}}^*_{\mathrm{CGF}}(S,E)$.

Proof. In CGF, the only reachable prime processes are defined in continuations by some identifiers (or the identifiers themselves). These are precisely those listed by $\widehat{\mathcal{P}}^*_{\mathrm{CGF}}$. ∎

Corollary 4.2. The set of prime processes of CGF system is always finite.

Proof. Clearly the only rule from the definition of prime approximation that could possibly cause recursion producing infinitely many new processes is (iii). However, since there is only a finitely many process identifiers used in an environment and finitely many channel names (as no restriction is allowed), this rule will be applied only to finitely many different arguments. ∎

Therefore the prime approximation gives us an efficient algorithm to enumerate prime processes of a CGF system – we will use this in our implementation.

In case of the full $\mathcal{S}\pi$, the enumeration of the set of prime processes of a system isn't so simple, as the set can be infinite (as we show later). However, we can still define a systematic way of obtaining an approximation.

Definition. Given an $\mathcal{S}\pi$ system (S, E), the *prime approximation* of (S, E) is $\widehat{\mathcal{P}}^*(S, E)$ defined as $\bigcup_{i=0}^{\infty} \widehat{\mathcal{P}}^i$ where $\widehat{\mathcal{P}}^*(S,E)$

$$\widehat{\mathcal{P}}^{(0)} = \{P_i : S \equiv \{\!|n_1 \times P_1, \dots, n_m \times P_m|\!\}^*\},$$
$$\widehat{\mathcal{P}}^{(i+1)} = \{Q_i : (R_1|R_2) \xrightarrow{\tau@r} Q,\ Q \equiv \{\!|k_1 \times Q_1, \dots, k_l \times Q_l|\!\}^*,\ R_1, R_2 \in \widehat{\mathcal{P}}_i\} \cup \widehat{\mathcal{P}}^i.$$

When modelling with ODEs, the solution is a set of real valued functions over time, one for each component of the model. Similarly, the continuous semantics will assign a real valued function over time to each prime process.

Definition. Given a system (S, E), for each $P \in \widehat{\mathcal{P}}(S, E)$, define the *quantity function of* P to be $[P] \colon \mathbb{R} \to \mathbb{R}$, a function that gives the (real-valued) quantity of the process P in the system at a given time. $[P]$

Communication in $\mathcal{S}\pi$ can essentially happen only between two prime processes. When prime processes P_1 and P_2 communicate, they evolve to Q_1 and Q_2 respectively, possibly bound under the same restriction. These are not necessarily prime, but can be decomposed into primes. Intuitively, if P_1 and P_2 produce P_3, the differential equation of P_3 should depend on P_1 and P_2. But it also should depend on the quantity of P_3 that is produced in the communication of P_1 and P_2 – we need the following operator.

Definition. Define the *mutliplicity operator*, $\# \colon \mathcal{P} \times \widehat{\mathcal{P}} \to \mathbb{N}$ as $S\#P = n$ if $S \equiv \{\!|n \times P, \dots|\!\}$ is the prime decomposition of S. $\cdot\#\cdot$

Note that for S and P in CGF, we can define $\#$ without considering prime decompositions

$$S\#Q = \begin{cases} 1 & \text{if } S = Q, \\ P_1\#Q + P_2\#Q & \text{if } S = P_1|P_2, \\ R\{\widetilde{x} \mapsto \widetilde{\psi}\}\#Q & \text{if } S = A\langle\widetilde{\psi}\rangle,\ A(\widetilde{x}) \overset{\text{def}}{=} R \in E, \\ 0 & \text{otherwise.} \end{cases}$$

This gives an efficient algorithm for computing $\#$; we will use it in our implementation.

Consider a system (S, E) and its evolution over time. The prime decomposition theorem tells us that each derivative of S will be a parallel composition of prime processes from the (possibly infinite) set $\widehat{\mathcal{P}}(S, E)$ and so can be represented as (possibly infinitely dimensional) vector. In the original discrete case, each transition from S to S' can be characterised by a change in the populations of the individual prime processes. This change will only depend on the population of the prime processes in S. We will try to explicitly characterise it.

There is only a limited number of ways a population of a process P can change. It can increase, as a result of communication between two prime processes, or as a result of a silent transition of a prime process (either caused by a silent action or internal communication). It can decrease as a result of communication with another prime process or due to a silent transition (again either caused by a silent action or internal communication). We will define multisets of *enter transitions* and *exit transitions* of a process (both in versions due to channel communication and silent transition). See Figure 4.1 for an overview.

Definition. The multiset of *enter channel transitions of a process* P *with respect to a system* (S, E) is $\mathrm{Enter}_{ch,S,E}$

$$\begin{aligned} \mathrm{Enter}_{ch,S,E}(P) = \{\!|n \times (a, R, T) : {} & R, T \in \widehat{\mathcal{P}}(S, E), \\ & R \xrightarrow{(\varphi)!a\langle\widetilde{\psi}\rangle} R',\ T \xrightarrow{?a\langle\widetilde{\psi}\rangle} T', \\ & (\mathrm{new}\,\varphi)(R'|T')\#P = n|\!\}. \end{aligned}$$

The multiset of *enter silent transitions of a process* P *with respect to a system* (S, E) is $\mathrm{Enter}_{\tau,S,E}$

$$\begin{aligned} \mathrm{Enter}_{\tau,S,E}(P) = \{\!|n \times (r, Q) : {} & Q \in \widehat{\mathcal{P}}(S, E), \\ & Q \xrightarrow{\tau@r} Q', \\ & Q'\#P = n|\!\}. \end{aligned}$$

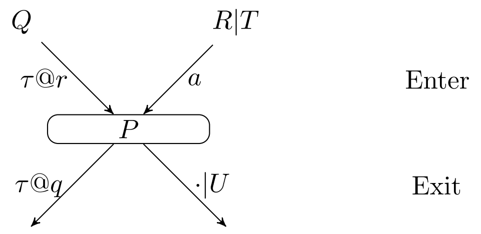


Figure 4.1: A diagram overviewing the four ways how a prime process can increase and decrease its population.

$\mathrm{Exit}_{ch,S,E}$ **Definition.** The multiset of *exit channel transitions of a process P with respect to a system* (S,E) is

$$\begin{aligned}\mathrm{Exit}_{ch,S,E}(P) = \{\!|(a,U) : U \in \widehat{\mathcal{P}}(S,E),\ P \overset{(\varphi)!a\langle\widetilde{\psi}\rangle}{\longrightarrow} P',\ U \overset{?a\langle\widetilde{\psi}\rangle}{\longrightarrow} U'|\!\}\uplus\\ \{\!|(a,U) : U \in \widehat{\mathcal{P}}(S,E),\ P \overset{?a\langle\widetilde{\psi}\rangle}{\longrightarrow} P',\ U \overset{(\varphi)!a\langle\widetilde{\psi}\rangle}{\longrightarrow} U'|\!\}.\end{aligned}$$

$\mathrm{Exit}_{\tau,S,E}$ The multiset of *exit delay transitions of a process P with respect to a system* (S,E) is

$$\mathrm{Exit}_{\tau,S,E}(P) = \{\!|r : P \overset{\tau@r}{\longrightarrow} P'|\!\}.$$

For processes in CGF, we can give an efficient enumeration of the above multisets. Because there is no restriction operator, the standard form of a CGF process is in fact its prime decomposition as well. Therefore we can use arguments similar to the Theorem 3.3 to given an algorithm for constructing the Enter and Exit multisets. We will use this in our implementation.

Consider a general $\mathcal{S}\pi$ system (S,E) and let $P_1,\dots,P_n$ be all the prime processes of $\widehat{\mathcal{P}}(S,E)$ (the assumption of finiteness only simplifies the following argument) and let $S\#P_i = n_i$. Say S undergoes a transition to S' and take a prime process, say P_1. If the multiset of enter channel transition of P_1 contains a triple (a,P_i,P_j), $P_1 \neq P_i \neq P_j \neq P_1$, the probability of the transition of $S \longrightarrow S'$ being due to communication of P_i and P_j is proportional to $r_a \cdot n_i \cdot n_j$ (by argument similar to the proof of Theorem 3.3 we can treat the case of when $P_i = P_j$ and also the enter silent transitions), resulting in a positive change in the population of P_1. Similarly, if we take (a,P_i), $P_i \neq P_1$, from the exit channel transition multiset of P_1, we get the probability of the transition being due to the communication of P_1 and P_i to be $r_a \cdot n_1 \cdot n_i$, resulting in a negative change of population of P_1.

The time the transition $S \longrightarrow S'$ takes is from an exponential distribution with parameter equal to the sum of these probabilities, say λ. This has a mean λ^{-1}. Informally, if we take a proportion of this time, δt, we can argue that the average population change of P_1 after time δt (denoted by $S(\delta t)\#P_1$) will be

$$\begin{aligned}S(\delta t)\#P_1 - S\#P_1 = &\sum_{(a,P_i,P_j)\in\mathrm{Enter}_{ch,S,E}(P)} r_a \cdot c(n_i,n_j)\delta t + \sum_{(r,P_i)\in\mathrm{Enter}_{\tau,S,E}(P)} r \cdot n_i\delta t\\ &- \sum_{(a,P_i)\in\mathrm{Exit}_{ch,S,E}(P)} r_a \cdot c(P_1,P_i)\delta t - \sum_{r\in\mathrm{Exit}_{\tau,S,E}(P)} r \times n_1\delta t,\end{aligned}$$

where $c(\cdot,\cdot)$ is the number of possible combinations, defined as

$$c(n_i,n_j) = \begin{cases} n_i \cdot n_j & \text{if } i \neq j,\\ n_i \cdot (n_i - 1) & \text{if } i = j.\end{cases}$$

Taking the limit of $\delta t \to 0$, we can get an intuitive notion of differential equation describing the evolution of P_1. In general, we arrive at the following definition.

Definition. The *system of differential equations of a system* (S, E) consists of the following, for each $P \in \widehat{\mathcal{P}}(S, E)$:

$$\begin{aligned}\frac{\mathrm{d}[P](t)}{\mathrm{d}t} = & \sum_{(a,R,T)\in \mathrm{Enter}_{ch,S,E}(P)} r_a \cdot c([R](t),[T](t)) + \sum_{(r,Q)\in \mathrm{Enter}_{\tau,S,E}(P)} r \cdot [Q](t) \\ & - \sum_{(a,U)\in \mathrm{Exit}_{ch,S,E}(P)} r_a \cdot c([P](t),[U](t)) - \sum_{r\in \mathrm{Exit}_{\tau,S,E}(P)} r \cdot [P](t),\end{aligned}$$

where $c(\cdot,\cdot)$ is the number of possible combinations, defined as

$$c([P](t),[Q](t)) = \begin{cases} [P](t)\cdot[Q](t) & \text{if } P \neq Q, \\ [P](t)\cdot([P](t)-1) & \text{if } P = Q. \end{cases}$$

Clearly, the initial conditions of the model are given by the top-level process of the system and so we can complete the definition of the continuous semantics of $\mathcal{S}\pi$ systems.

Definition. The *continuous semantics of a system* (S, E) is

$$\{[P] : P \in \widehat{\mathcal{P}}(S, E)\}$$

such that the quantity functions $[P]$ satisfy the system of differential equations of (S, E) with initial conditions given by

$$[P](0) = S\#P.$$

Example 9. Consider the CGF environment E_{lotka}

$$\begin{aligned} Prey &\overset{\mathrm{def}}{=} \tau@r_{reproduce}.(Prey|Prey) + ?eat, \\ Predator &\overset{\mathrm{def}}{=} !eat.(Predator|Predator) + \tau@r_{die} \end{aligned}$$

with an initial process

$$System = 1000 \times Prey | 950 \times Predator.$$

We have argued in the previous chapter how this can represent a simple predator-prey model. The set of prime processes of *System* is (possibly obtained as $\widehat{\mathcal{P}}^*_{\mathrm{CGF}}(System, E_{lotka})$ by observing that all the processes can occur)

$$\widehat{\mathcal{P}}(System, E_{lotka}) = \{Prey, Predator\}.$$

We also have

$$\begin{aligned} \mathrm{Enter}_{ch,System,E_{lotka}}(Prey) &= \{\!|\,|\!\}, \\ \mathrm{Enter}_{\tau,System,E_{lotka}}(Prey) &= \{\!|2 \times (r_{reproduce}, Prey)|\!\}, \\ \mathrm{Exit}_{ch,System,E_{lotka}}(Prey) &= \{\!|(eat, Predator)|\!\}, \\ \mathrm{Exit}_{\tau,System,E_{lotka}}(Prey) &= \{\!|r_{reproduce}|\!\}, \\ \mathrm{Enter}_{ch,System,E_{lotka}}(Predator) &= \{\!|2 \times (eat, Predator, Prey)|\!\}, \\ \mathrm{Enter}_{\tau,System,E_{lotka}}(Predator) &= \{\!|\,|\!\}, \\ \mathrm{Exit}_{ch,System,E_{lotka}}(Predator) &= \{\!|(eat, Prey)|\!\}, \\ \mathrm{Exit}_{\tau,System,E_{lotka}}(Predator) &= \{\!|r_{die}|\!\}. \end{aligned}$$

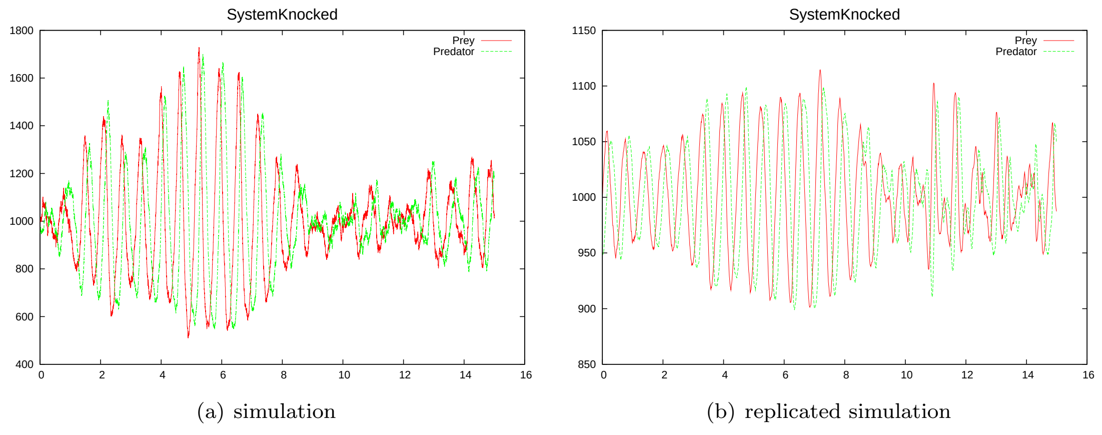
We get the set of differential equations of $(System, E_{lotka})$ to be

$$\begin{aligned} \frac{\mathrm{d}[Prey](t)}{\mathrm{d}t} &= r_{reproduce}\cdot[Prey](t) - r_{eat}\cdot[Predator](t)\cdot[Prey](t), \\ \frac{\mathrm{d}[Predator](t)}{\mathrm{d}t} &= r_{eat}\cdot[Predator](t)\cdot[Prey](t) - r_{die}\cdot[Predator](t). \end{aligned}$$

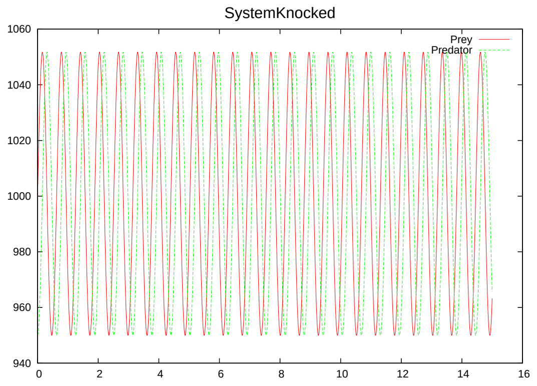
The initial conditions for the process *System* are

$$[Prey](0) = 1000, \qquad [Predator](0) = 950.$$

and hence the continuous semantics of $(System, E_{lotka})$ is the solution to the system of ODEs of $(System, E_{lotka})$ with these initial conditions. See Figure 4.2 for a numerical solution to this, compared to a sample simulation trace and an average of multiple simulations. Also note how the continuous semantics exactly corresponds to the well-known Lotka-Volterra equations modelling the same situation directly with ODEs (for example see [17]).



(a) simulation (b) replicated simulation



(c) ODE solution

Figure 4.2: Simulation and ODE solution of the process *SystemKnocked* from the environment E_{lotka}.

We now state a Theorem analogous to Theorem 3.1, saying that the structural congruence respects the continuous semantics.

Theorem 4.3. If (S_1, E) and (S_2, E) are two $\mathcal{S}\pi$ systems such that $S_1 \equiv S_2$, then there exists a bijection $\phi\colon \widehat{\mathcal{P}}(S_1, E) \to \widehat{\mathcal{P}}(S_2, E)$ such that for every prime process $P \in \widehat{\mathcal{P}}(S_1, E)$, $[P] = [\phi(P)]$.

Proof. The proof is due to the fact that in constructing the sets of prime processes and enter and exit multisets, we consider processes modulo structural congruence. ■

4.1 Translation to CGF

The definition of prime processes of a general $\mathcal{S}\pi$ system cannot be made so explicit as for the case of CGF. We first present an example demonstrating how some of the functionality of the restriction operator can be "emulated" in CGF.

Example 10. Consider the environment E_{HCl}

$$\begin{aligned} H &\stackrel{\text{def}}{=} (\text{new}\, e@10.0)(!share\langle e\rangle.H\_b\langle e\rangle), \\ H\_b(e) &\stackrel{\text{def}}{=} !e.H, \\ Cl &\stackrel{\text{def}}{=} ?share(e).Cl\_b\langle e\rangle, \\ Cl\_b(e) &\stackrel{\text{def}}{=} ?e.Cl. \end{aligned}$$

In a process consisting of parallel composition of processes Cl and H, i.e. $\{\!|n \times Cl, m \times H|\!\}$, the only possible reaction that can occur is a communication between H and Cl on the *share* channel. This gives rise to a complex (let us call it HCl),

$$(\text{new}\, e@10.0)(!e.H|?e.Cl).$$

This process cannot react with any other process and has only one possible transition to the process $(\text{new}\, e@10.0)(H|Cl)$, which is structurally congruent to $H|Cl$. This suggests an "equivalent" process $\tau@10.0.(H|Cl)$ and thus a CGF environment $E_{HCl'}$

$$\begin{aligned} H' &\stackrel{\text{def}}{=} !share.HCl', \\ Cl' &\stackrel{\text{def}}{=} ?share, \\ HCl' &\stackrel{\text{def}}{=} \tau@10.0.(H'|Cl'). \end{aligned}$$

Now the resulting CTMC (see Figure 4.3) from a process $S = n \times H|m \times Cl$ is the same as from $S' = n \times H'|m \times Cl'$ (it can be easily observed that the CTMC will be same as that from Figure 4.3 with H replaced by H' and Cl by Cl').

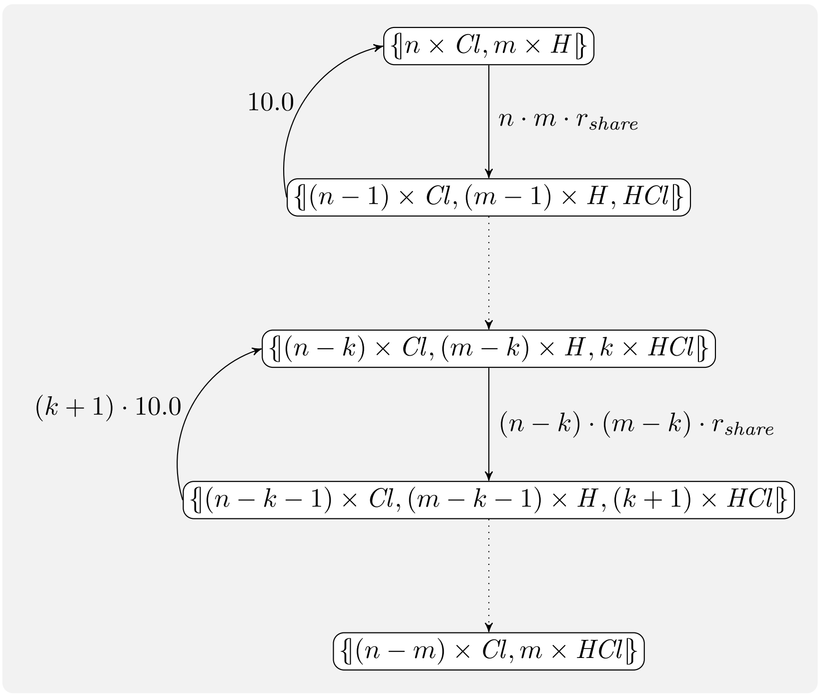

$\{\!|n \times Cl, m \times H|\!\}$
10.0
$n \cdot m \cdot r_{share}$
$\{\!|(n-1) \times Cl, (m-1) \times H, HCl|\!\}$
$\{\!|(n-k) \times Cl, (m-k) \times H, k \times HCl|\!\}$
$(k+1) \cdot 10.0$
$(n-k) \cdot (m-k) \cdot r_{share}$
$\{\!|(n-k-1) \times Cl, (m-k-1) \times H, (k+1) \times HCl|\!\}$
$\{\!|(n-m) \times Cl, m \times HCl|\!\}$

Figure 4.3: CTMC for E_{HCl}.

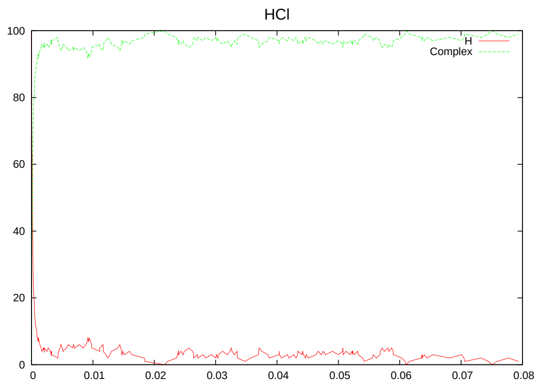
The continuous semantics $(S', E_{HCl'})$ is

$$\frac{\mathrm{d}[H'](t)}{\mathrm{d}t} = 10.0 \cdot [HCl'](t) - r_{share} \cdot [H'](t) \cdot [Cl'](t), \tag{4.1}$$

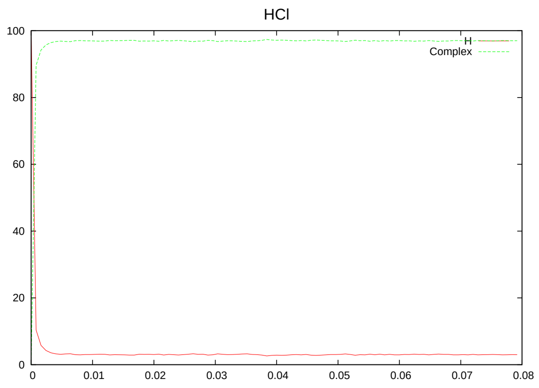
$$\frac{\mathrm{d}[Cl'](t)}{\mathrm{d}t} = 10.0 \cdot [HCl'](t) - r_{share} \cdot [H'](t) \cdot [Cl'](t), \tag{4.2}$$

$$\frac{\mathrm{d}[HCl'](t)}{\mathrm{d}t} = r_{share} \cdot [H'](t) \cdot [Cl'](t) - 10.0 \cdot [HCl'](t). \tag{4.3}$$

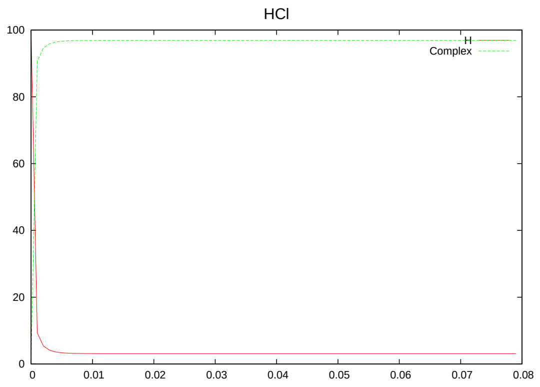
See Figure 4.4 for a solution to the induced system of ODEs and an example of a simulation.



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure 4.4: Sample simulation of the HCl model and a numerical solution to the system of ODEs generated by the continuous semantics.

The above can be trivially compared with the continuous semantics of the system (S, E_{HCl}) where

$$S = n \times H | m \times Cl.$$

We have $\widehat{\mathcal{P}}(S, E_{HCl}) = \{H, Cl, (\text{new}\, e@10.0)(Cl\_b\langle e\rangle | H\_b\langle e\rangle)\}$. The system of ODE's for (S, E_{HCl}) then is

$$\frac{\mathrm{d}[H](t)}{\mathrm{d}t} = 10.0 \cdot [(\text{new}\, e@10.0)(Cl\_b\langle e\rangle | H\_b\langle e\rangle)](t) - r_{share} \cdot [H](t) \cdot [Cl](t),$$

$$\frac{\mathrm{d}[Cl](t)}{\mathrm{d}t} = 10.0 \cdot [(\text{new}\, e@10.0)(Cl\_b\langle e\rangle | H\_b\langle e\rangle)](t) - r_{share} \cdot [H](t) \cdot [Cl](t),$$

$$\frac{\mathrm{d}[(\text{new}\dots)(Cl\_b\langle e\rangle | H\_b\langle e\rangle)](t)}{\mathrm{d}t} = r_{share} \cdot [H](t) \cdot [Cl](t) - 10.0 \cdot [(\text{new}\, e@10.0)(Cl\_b\langle e\rangle | H\_b\langle e\rangle)](t).$$

Compare this with the ODEs generated from the CGF translation of E_{HCl} (equations (4.1) - (4.3)).

4.2 Summary

We defined continuous semantics for the full $\mathcal{S}\pi$. We have shown efficient algorithms for generating the set of ODEs for a system in CGF – this will form the basis of our implementation of the continuous semantics. We gave an example showing that the new operator can be sometimes "emulated" in CGF.

5 Finiteness conditions and convergence investigations

We start this chapter with two examples of $\mathcal{S}\pi$ systems where the continuous semantics gives an infinite set of ODEs.

Example 11. Consider a simple model from [8]. We won't give the biological details here; it suffices to say that the model concerns polymerization of *actin* (a basic unit of cytoskeletal networks in eukaryotic cells) monomers into filaments. We take the simplest model from [8], where the monomers (processes Af below) can form linear chains. In such process, the chain is able to bind a new monomer on one end and to disassociate its monomers on the other. We have the environment E_{actin} containing the defining equations

$$\begin{aligned}
Af &\overset{\text{def}}{=} (\text{new}\, r@p)(?c(l).Al\langle l\rangle + !c\langle r\rangle.Ar\langle r\rangle),\\
Al(l) &\overset{\text{def}}{=} (\text{new}\, r@p)(!l.Af + !c\langle r\rangle.Ab\langle l, r\rangle),\\
Ar(r) &\overset{\text{def}}{=} ?r.Af,\\
Ab(l,r) &\overset{\text{def}}{=} !l.Ar\langle r\rangle
\end{aligned}$$

and a top-level process $S = n{\times}Af$. Two Af processes can create a complex $(\text{new}\, r_1@p)(Ar\langle r_1\rangle|Al\langle r_1\rangle)$ which can then bind another Af process (the Al subprocess) or release an Af process (the Ar) subprocess.

To get all the prime processes of this system, observe that

$$\begin{aligned}
\widehat{\mathcal{P}}^{(0)} &= \{Af\},\\
\widehat{\mathcal{P}}^{(1)} &= \widehat{\mathcal{P}}^{(0)} \cup \{(\text{new}\, r_1@p)(Ar\langle r_1\rangle|Al\langle r_1\rangle)\},\\
\widehat{\mathcal{P}}^{(2)} &= \widehat{\mathcal{P}}^{(1)} \cup \{(\text{new}\, r_1@p, r_2@p)(Ar\langle r_1\rangle|Ab\langle r_1, r_2\rangle|Al\langle r_2\rangle)\},\\
\widehat{\mathcal{P}}^{(k+1)} &= \widehat{\mathcal{P}}^{(k)} \cup \{\underbrace{(\text{new}\, r_1@p, \ldots, r_{k+2}@p)(Ar\langle r_1\rangle|Ab\langle r_1, r_2\rangle|Ab\langle r_2, r_3\rangle|\cdots|Ab\langle r_{k+1}, r_{k+2}\rangle|Al\langle r_{k+2}\rangle)}_{A_{k+2}}\}.
\end{aligned}$$

since $(A_{k+1}|Af) \xrightarrow{\tau@r_c} A_{k+2}$ and $A_{k+1} \xrightarrow{\tau@p} A_k$ are the only possible reactions involving processes in $\widehat{\mathcal{P}}^{(k)}$. Therefore $\widehat{\mathcal{P}}^*(S, E_{actin}) = \{Af, A_1, A_2, \ldots, A_{n-2}\} = \widehat{\mathcal{P}}(S, E)$ and the system of ODEs is then

$$\begin{aligned}
\frac{\mathrm{d}[Af](t)}{\mathrm{d}t} &= 2 \cdot p \cdot [A_1](t) + \sum_{k=2}^{n-2}(p \cdot [A_k](t) - r_c \cdot [A_k](t)) - r_c \cdot [Af](t)([Af](t) - 1),\\
\frac{\mathrm{d}[A_1](t)}{\mathrm{d}t} &= r_c \cdot [Af](t) \cdot ([Af](t) - 1) + p \cdot [A_2](t) - p \cdot [A_1](t),\\
&\vdots\\
\frac{\mathrm{d}[A_k](t)}{\mathrm{d}t} &= r_c \cdot [Af](t)[A_{k-1}](t) + p \cdot [A_{k+1}](t) - p \cdot [A_k](t),\\
&\vdots\\
\frac{\mathrm{d}[A_{n-2}](t)}{\mathrm{d}t} &= r_c \cdot [Af](t)[A_{n-3}](t) + p \cdot [A_{n-1}](t) - p \cdot [A_{n-2}](t),
\end{aligned}$$

The above system of ODEs has finitely many equations. However, their number (and the length of the summation in the first one) depends on the number of Af processes in the top-level process. Moreover, in [8], the authors give more complicated systems modelling branching of the actin chains – this results in a combinatorial explosion of the number of prime processes (which can be equal to the number of different binary trees with n nodes). Finally, in case the environment additionally contains a process that generates new Af processes, say a defining equation of the form

$$F \stackrel{\text{def}}{=} \tau@1.0.(F|Af)$$

the resulting set of ODEs is (countably) infinite, with the first ODE containing an infinite summation.

Example 12. Take the environment E_{star} containing defining equations

$$\begin{aligned}
P &\stackrel{\text{def}}{=} (\text{new}\, e@r)(!s\langle e\rangle|?e),\\
Q &\stackrel{\text{def}}{=} ?s(x).F\langle x\rangle,\\
F(x) &\stackrel{\text{def}}{=} \tau@r.(!e|F\langle x\rangle)
\end{aligned}$$

and a top-level process $S = P|Q$. The prime processes of (S, E_{star}) are P, Q and

$$\begin{aligned}
(\text{new}\, e@r)(?e|F\langle e\rangle) &= P_0,\\
(\text{new}\, e@r)(?e|!e|F\langle e\rangle) &= P_1,\\
&\vdots\\
(\text{new}\, e@r)(?e|\underbrace{!e|\cdots|!e}_{k}|F\langle e\rangle) &= P_k,\\
&\vdots
\end{aligned}$$

This set is (countably) infinite.

The above examples show $\mathcal{S}\pi$ systems that have continuous semantics producing an infinite set of differential equations. We will try to reason about this property further. We formally define two notions of what it means for a system to be finite, based on the intuition from the above examples. We show that if a system is finite, we can replace it with an equivalent CGF system.

We then formulate two conditions that will guarantee these notions. One will be based on restricting the syntax of $\mathcal{S}\pi$, not allowing parallel composition in continuations (we will call this subset of $\mathcal{S}\pi$ the *non-productive* $\mathcal{S}\pi$, np$\mathcal{S}\pi$). The other is based on static analysis of what can happen to newly generated channel names.

Using these conditions, we translate several finitely scalable models from SPiM and try to investigate the relationship between their discrete and continuous semantics.

5.1 Conditions for finiteness

As was shown in the example above, a continuous semantics of a process does not always have to be a finite system of differential equations. We first specify what exactly it means for a continuous semantics to give a finite set of ODEs. We give two possible definitions. The first one, which we will call just finiteness, assumes a fixed system (S, E) and requires a finite number of processes. The second is stronger and requires not only a fixed system (S, E) to give finitely many prime processes, but also to make the number of prime processes independent of *scaling*, that is taking $n \times S$ instead of S for some n.

Definition. We say that a system (S, E) is *finite* if the set of prime processes of S with respect to E, $\widehat{\mathcal{P}}(S, E)$ is finite.

Example 13. Any CGF system is finite, as we have proved in the Corollary 4.2.

Definition. We say that a system (S, E) is *scalably finite* if there exists a finite set of prime processes $\mathcal{P}^*$ such that the prime processes of $(n \times S, E)$ is a subset of $\mathcal{P}^*$.

Example 14. Any CGF system (S, E) is scalably finite – the prime approximation $\widehat{\mathcal{P}}^*_{CGF}(S, E)$ does not depend on multiplicities of processes inside S. We have shown that the prime approximation is always finite and so can be taken as the set $\widehat{\mathcal{P}}^*$ in the previous definition.

Clearly if a system is scalably finite, then it is also finite. The opposite is not true – the system from the actin example is finite but not scalably finite.

The following theorem shows why it is useful to consider conditions for finiteness – if a system is finite, then we can replace it with a CGF system. This will allow modelling of the system in JSPiM.

Theorem 5.1. If a system (S, E) is finite, then there exists a CGF system (S', E') with the same continuous semantics.

Proof. Assume (S, E) is finite. Then there are finitely many prime processes in $Ds(S) = \{P_1, \ldots, P_n\}$. Take a set of identifiers $\mathcal{I} = \{A_1, \ldots, A_n\}$. We will define a translation function $\theta \colon \mathcal{P} \to \mathcal{P}_{\mathrm{CGF}}$ to define a CGF environment $E' = \{A_i \overset{\mathrm{def}}{=} \theta(P_i) : i = 1, \ldots, n\}$ and a top-level process $S' = \theta(S)$ with the same continuous semantics as (S, E). For P not a prime, that is P with prime decomposition $\{\!|k_1 \times Q_1, \ldots, k_m \times Q_m|\!\}$, define $\theta(P) = \{\!|k_1 \times \theta(Q_1), \ldots, k_m \times \theta(Q_m)|\!\}$. For P prime, consider the sets $\mathrm{Exit}_{ch,S,E}(P)$ and $\mathrm{Exit}_{\tau,S,E}(P)$. These are clearly both finite. We will define $\theta(P)$ to be a summation. Let $\mathrm{Exit}_{ch,S,E}(P)$ contains precisely m copies of (a, Q), corresponding to pairs of transitions $P|Q \xrightarrow{\tau@r_a} R_i$, $i = 1, \ldots, m$, where P does an output action. For transition i, define a new channel $a_{P,Q,i}$ where $i = 1, \ldots, m$, with the same rate as a and add a summand $!a_{P,Q,i}.\theta(R_i)$. If $\mathrm{Exit}_{ch,S,E}(P)$ contains precisely l copies of (a, Q), corresponding to transitions $P|Q \xrightarrow{\tau@r_a} T_j$, $j = 1, \ldots, l$, where P does an input action, add summands $?a_{Q,P,j}$. Similarly for r in $\mathrm{Exit}_{\tau,S,E}(P)$ corresponding to a silent transition $P \xrightarrow{\tau@r} R$, add summand $\tau@r.\theta(R)$.

Now clearly the identifier instances $A_1, \ldots, A_n$ are the only prime processes in (S', E'). By the above construction θ defines a multiset bijection between $\mathrm{Exit}_{ch,S,E}(P_j)$ and $\mathrm{Exit}_{ch,S',E'}(A_j)$, that is $(a, P, Q) \in \mathrm{Exit}_{ch,S,E}(P_j)$ iff $(a_{P,Q,i}, \theta(P), \theta(Q)) \in \mathrm{Exit}_{ch,S',E'}(A_j)$ for some i – and so the negative terms in the ODE associated with $[A_j]$ are the same as the negative terms in those for $[P_j]$. Similarly for $\mathrm{Exit}_{\tau,S,E}(P_j)$ and $\mathrm{Exit}_{\tau,S',E'}(A_j)$. Also, it is easy to see that θ defines a bijection between the Enter multisets of P_j and A_j and so the positive terms in the ODEs for $[A_j]$ and $[P_j]$ are the same and so (S', E') and (S, E) have the same continuous semantics. ■

5.1.1 Syntactic restriction

We define a syntactical restriction of $\mathcal{S}\pi$ that will guarantee finiteness. An obvious choice for this restriction is to forbid parallel compositions in continuations. This will result in the standard form of any derivative of such process having only finitely many parallel components.

Definition. *Non-productive π calculus* (np$\mathcal{S}\pi$) is a subset of stochastic π calculus which allows parallel composition only at the levels of top-level processes, that is processes of np$\mathcal{S}\pi$ are the set $\mathcal{P}_{\text{np}}$ minimal such that $\mathbf{0} \in \mathcal{P}_{\text{np}}$, $\sum_{i\in I} \alpha_i.P_i \in \mathcal{P}_{\text{np}}$ if $P_i \in \mathcal{P}_{\text{np}}$ for all $i \in I$, $A\langle\widetilde{\psi}\rangle \in \mathcal{P}_{\text{np}}$ and $(\text{new}\, a@r)P \in \mathcal{P}_{\text{np}}$ if $P \in \mathcal{P}_{\text{np}}$. An environment of np$\mathcal{S}\pi$ then contains defining equations of the form

$$A(\widetilde{x}) \stackrel{\text{def}}{=} P$$

where $P \in \mathcal{P}_{\text{np}}$. We allow parallel composition of identifier instances in the top-level processes, which then become $A_1\langle\widetilde{\psi_1}\rangle|A_2\langle\widetilde{\psi_2}\rangle|\cdots|A_m\langle\widetilde{\psi_m}\rangle$ where $\widetilde{\psi_i} \in \mathcal{N}^{n_{A_i}}$.

Proposition 5.2. Let (S, E) be a system of np$\mathcal{S}\pi$. Then the continuous semantics of (S, E) is finite.

Proof. Let S have a standard form $(\text{new}\,\varphi)\{\!|n_1 \times P_1, \dots, n_m \times P_m|\!\}$. Take any derivative of S, say S'. Let the standard form of S' be $(\text{new}\,\varphi')\{\!|k_1 \times Q_1, \dots, k_l \times Q_l|\!\}$. It is easy to see, since no parallel composition is allowed after continuations, that the sum of n_i is greater or equal to the sum of k_j. Therefore there exists a constant N (for example the sum of all n_i) which bounds the sum of k_j. Also, each Q_j is a summation or an identifier instance and so has to be a subprocess of a right hand side of some defining equation in E. Thus there must be a constant K which bounds the number of free variables of all Q_j's in all S'. Therefore there is a constant (e.g. $N \cdot K$) which bounds the number of free variables of $\{\!|k_1 \times Q_1, \dots, k_l \times Q_l|\!\}$ in *all* possible S'. Therefore there is always a standard form of S' which has φ' such that $|\varphi'| < N \cdot K$. Then there is only finitely many combinations in which the different Q_j's can be grouped according to shared variables and so there is only finitely many prime processes that can arise within a derivative of S. ∎

Unfortunately, this restriction is too strict and none of the non-CGF models in the collection in Appendix B belong to np$\mathcal{S}\pi$.

Moreover, np$\mathcal{S}\pi$ does not guarantee scalable finiteness.

Proposition 5.3. The continuous semantics of (S, E) is not necessarily scalably finite.

Proof. Consider the environment E_{actin} – clearly this is an environment of $\mathcal{P}_{\text{np}}$ and a top-level process $S = n \times Af$. Then, as shown in an example above, the size of $\widehat{\mathcal{P}}(S, E)$ is n. ∎

5.1.2 Restriction on private names

We will informally argue that the two examples from the end of the previous chapter suggest the only two possible ways a system can become infinite. In the first example, the complications were caused by the processes $Ab\langle r_i, r_k\rangle$, which were "holding the restrictions together". In the original model, the number of these depended on the initial populations in the top-level process of the system – the system stayed finite, but not scalably finite. In the modification, this was made worse by the added process being able to create infinitely many of these "chaining" processes.

In the second example, the "infiniteness" was caused by a process creating arbitrary many copies of the private channel name, resulting in complexes of arbitrary "size".

We give a condition that eliminates the above scenarios. To eliminate the first case, we do not allow a process to "bind" two names potentially "originating in restrictions". We need to define what we mean by "binding" and "originating in a restriction".

We will say that a process *binds* two names or variables if they occur within the same summation or as parameters of an identifier instance. For example, the process $!a{+}?c(x).?b.!x$ binds a and b and so does $A\langle a, b\rangle$, but the process $!a|?b$ does not. In short, a process P binds a and b if it contains a and b as free names and there exist no processes Q and R such that $P \equiv Q|R$ where a is not free in Q and b is not free in R. This implies that if we have $(\text{new}\{a, b\})P$, we cannot find Q, R such that this would be congruent to $(\text{new}\, a)Q|(\text{new}\, b)R$.

For each variable in the definition of an environment, we need to decide if it can *originate in a restriction*, that is the input action defining the variable can receive (in any derivative of the top-level process) a name that comes from a restriction or the process identifier defining the variable can be instantiated with a name that comes from a restriction. For example, in the environment E_{actin}, the variable l in the first summand in the definition of Af can come from a restriction because a private name from an Af process can be shared on the channel c. Similarly, the variable l in the definition of $Al(l) \stackrel{\text{def}}{=}$ can come from a restriction, since Al gets instantiated in Af with a variable that can come from a restriction. A name originates in a restriction if it is bound by a restriction – for example the name r in the definition of Af in E_{actin} originates in a restriction. Also the r in $Ab\langle l,r\rangle$ in the definition of Al does and therefore $Ab\langle l,r\rangle$ can bind two names coming from a restriction – precisely what we want to avoid.

To eliminate the second case, we do not allow recursive "copying" of names and variables that can originate in a restriction.

Now assume that the two requirements are satisfied, that is, there is no process anywhere in the derivatives of a top-level process S that can potentially bind two names originating in restrictions and that there is a limit to the number of times a name from a restriction can be copied. We can take any derivative of S, say S' and consider its standard form $(\text{new}\,\varphi)\{\!|n_1 \times P_1, \ldots, n_m \times P_m|\!\}$. The names in $\varphi = \{e_1, \ldots, e_k\}$ are the only names originating in restrictions. Because of the first condition, the standard form can be split into k prime processes, each a restriction on one of the names from φ, containing some of P_i, say a process $E_i = (\text{new}\,e_i)\{\!|n_{i_1} \times P_{i_1}, \ldots, n_{i_l} \times P_{i_l}|\!\}$, $i = 1, \ldots, k$. Because of the second condition, there is a limit to the number of copies of e_i and therefore there is a limit to the sum of n_{i_j}. Because the P_{i_j} have to come from the definition of the environment which is finite, there is only a finite number of such E_i in all the derivatives S' and so (S,E) is finite. Because no part of our argument used the initial top-level process S, we also get (S,E) to be scalably finite.

We can use the above reasoning to see that the system from the model B.1 is scalably finite, as there is no copying of new names and no process binds two different names originating in restrictions. See B.2 for a translation of this to CGF (using the algorithm from the proof of Theorem 5.1).

We can also look at the system in the model B.5. This does not satisfy the above condition in its current form, since for example the process

$$E2 \stackrel{\text{def}}{=} (\text{new}\{k1@rk, d1@rd)(!a1\langle d1, k1\rangle.(?d1.E1+?k1.E2))$$

binds the two names $k1$ and $d1$ originating restrictions. However, we can see that these two names originate in the "same" restriction. We can observe that every name binding is like this and therefore conclude that the system is scalably finite and give its translation to CGF, the model B.6. Perhaps this example justifies why didn't give a formal definition of the above condition – to make it useful we would have needed to consider many special cases.

5.2 Relationship between continuous and discrete semantics

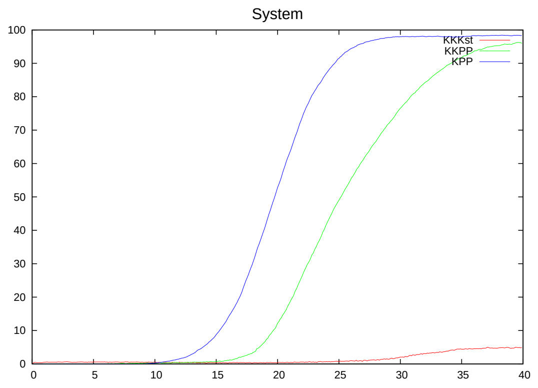
We will experiment with the continuous and discrete semantics by looking at available models for stochastic π calculus. In particular, we investigate whether some of the convergence properties from PEPA and BioPEPA hold for $\mathcal{S}\pi$.

In BioPEPA, the continuous concentration associated with each species is discretised into a number of levels. One can increase this granularity, resulting in "finer" state space of the underlying CTMC. It has been shown in [14] that the set of ODEs derived from the syntax of the model (which is not dependent on the number of levels in the case of BioPEPA) captures the limiting behaviour of the CTMC representing the discretised system.

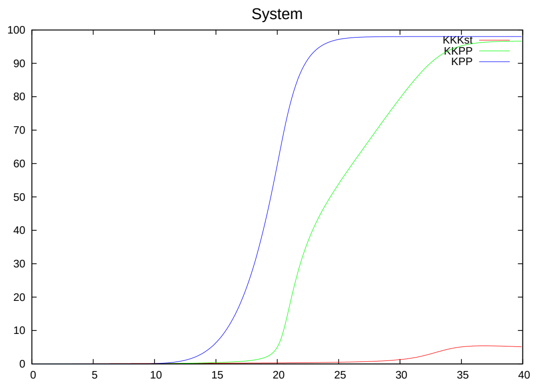
In $\mathcal{S}\pi$, there is no obvious "granularity" that can be increased. However, we can still try to investigate convergence in some sense. Each process can have an integer population in a system, which reaches a certain maximum over a time scale we can be interested in. If we divide the populations by this maximum, we can think of them as concentrations. Then clearly the larger this maximum is, the more "granularity" we get. Given a system (S,E), we can try to *scale* it, that is consider systems $(n \times S, E)$ for different n. If (S,E) is scalably finite, the systems $(n \times S, E)$ will have the same prime processes as $(n \times S, E)$ and therefore continuous semantics with the same prime processes. It may be of interest to investigate how the scaling influences the relationship between the discrete and continuous semantics. In particular, we will look at how the *average* of replicated simulation traces compares to the ODE solution when n is increased. Intuitively, this should result in a similar effect on the CTMC as the increased granularity – with larger populations of processes, the reaction rates increase and the state space of the CTMC becomes "finer". As opposed to BioPEPA, the set of ODEs from continuous semantics of $\mathcal{S}\pi$ *does* depend on the scaling. Therefore, for the comparison to make any sense, we need to always compare simulations and ODEs from the same scale.

In the following examples, we take models from the collection in Appendix B and experiment with the scaling. If the model is not in CGF, we use the ideas from the previous section and translate it (we have done this for the models B.1 and B.5). For each of the models, we take the initial system (S, E) and then scale it twice by a factor of ten, i.e. get systems $(10 \times S, E)$ and $(100 \times S, E)$. For each of these, we obtain an average and variance of 20 simulations traces. We will compare the averages with the respective ODE solutions. We will also look at the variance – we plot changing standard deviation over the time, scaled by the mean to allow comparison between the different scales. We show examples where the correspondence between the simulations and ODE solutions clearly shows and so we can expect the convergence to hold. However, we show an example where no such correspondence appears and so we can expect that the convergence does not hold in general for $\mathcal{S}\pi$ systems.

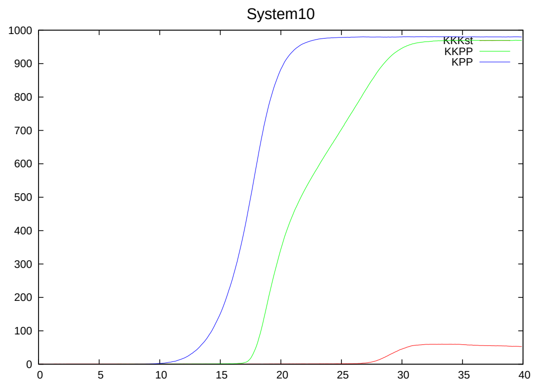
Example 15. In this example, we take a model which studies ultra-sensitivity of the mitogen-activated protein kinase (MAPK) cascade, as described in [21] and translated to stochastic π calculus in [31]. See Appendix B.5 for JSPiM description of the model. Originally, the model is in $\mathcal{S}\pi$. However, using the intuition from the previous section, we were able to translate it to CGF and therefore use JSPiM to analyse the continuous semantics. See Appendix B.6 for the translated model. In this case, the convergence of the discrete semantics is evident – the simulation average gets closer to the corresponding ODE solution as the scale increases. See Figure 5.1. Also, the variance decreases with increased scale, see Figure 5.2.



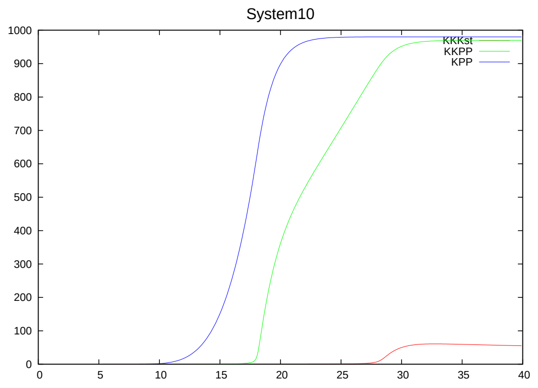
(a) Replicated simulation for the original model



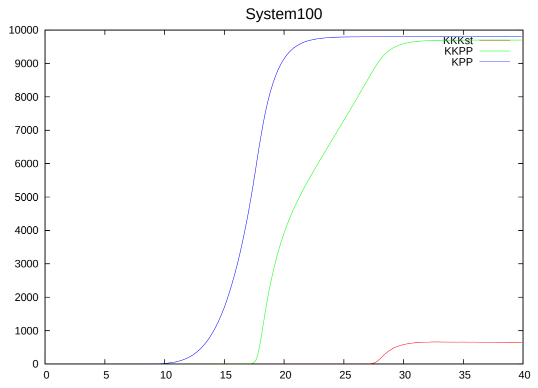
(b) ODE solution for the original model



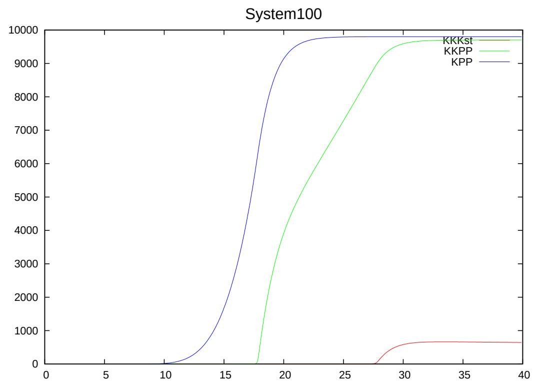
(c) Replicated simulation for the model scaled by 10



(d) ODE solution for the model scaled by 10

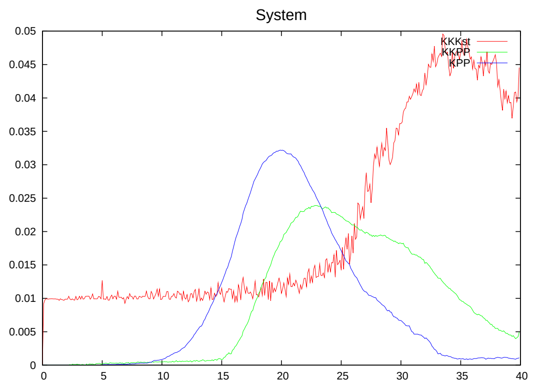


(e) Replicated simulation for the model scaled by 100

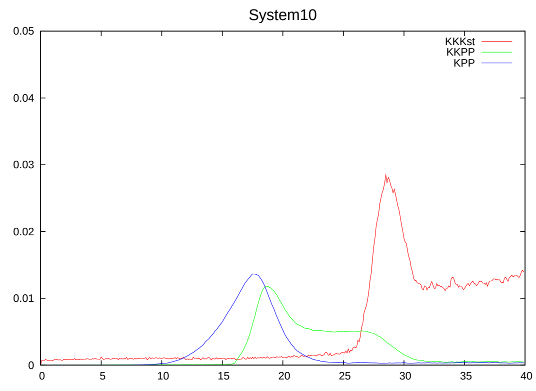


(f) ODE solution for the model scaled by 100

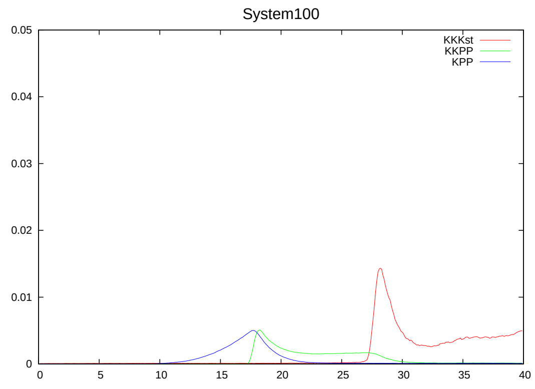
Figure 5.1: The effect of scaling on the simulation of the MAPK model. The average of 20 different runs of the simulation is compared to the solution of the corresponding ODEs for the original model and the original model scaled by factors of 10 and 100 respectively. We can expect the simulation averages to converge to the ODE solution as the scale increases.



(a) original model



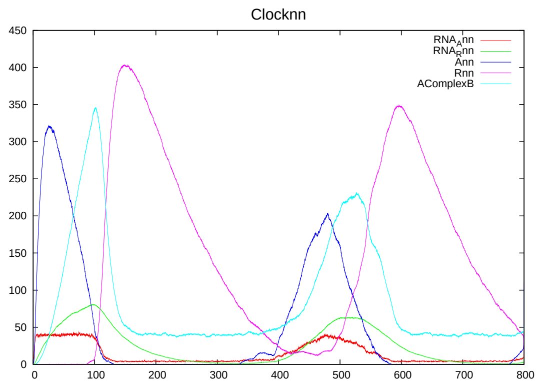
(b) scaled by 10



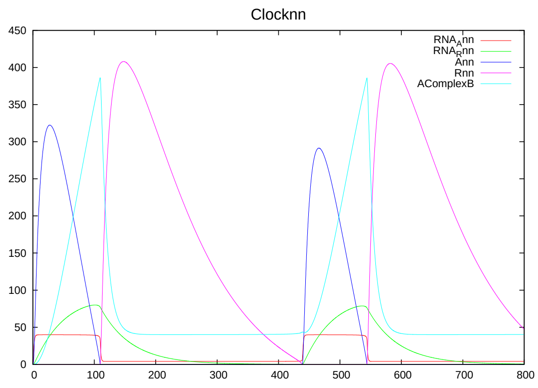
(c) scaled by 100

Figure 5.2: Plots of the changing variance for the MAPK model. For each scale, the standard deviation scaled by the maximal mean is plotted. We can see that the variance decreases as the scale increases.

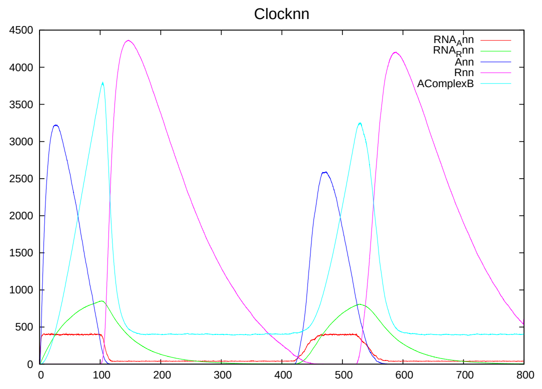
Example 16. We consider a model of circadian clock as defined in [30]. See Appendix B.1 for the original model and Appendix B.2 for the translation to CGF that we used for analysing the continuous semantics. Similar to the MAPK model, this model suggests the convergence of scaled simulation towards the scaled ODE solution. See Figure 5.3. In this case, the decrease in variance is not as significant as in the MAPK model. See Figure 5.4.



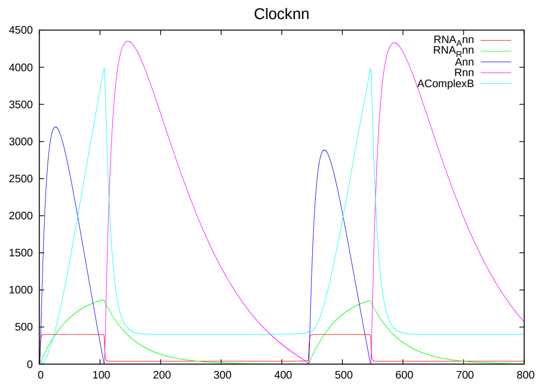
(a) Replicated simulation for the original system



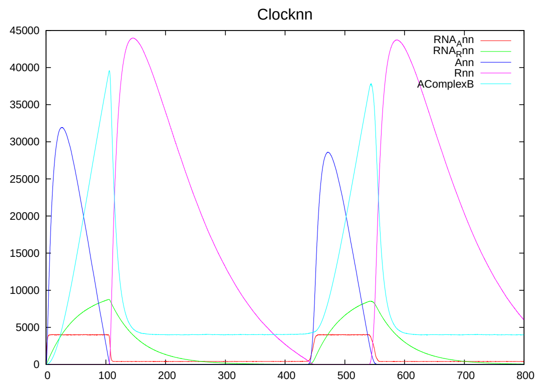
(b) ODE solution for the original system



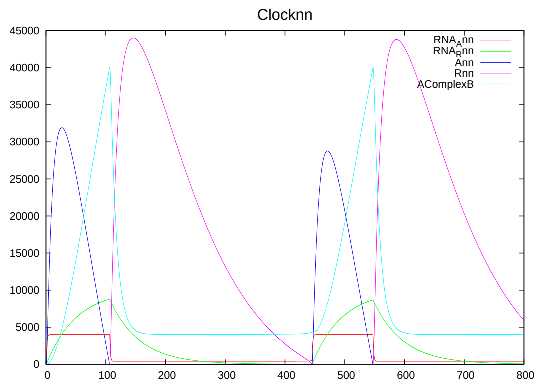
(c) Replicated simulation for the system scaled by 10



(d) ODE solution for the system scaled by 10

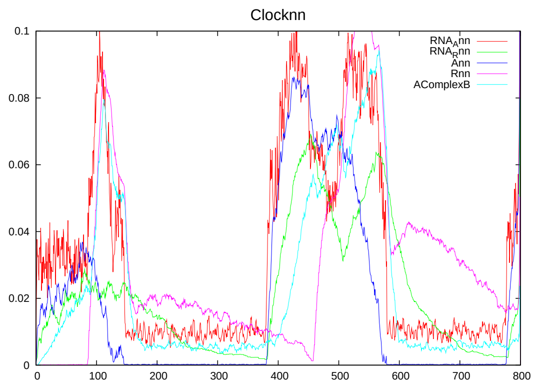


(e) Replicated simulation for the system scaled by 100

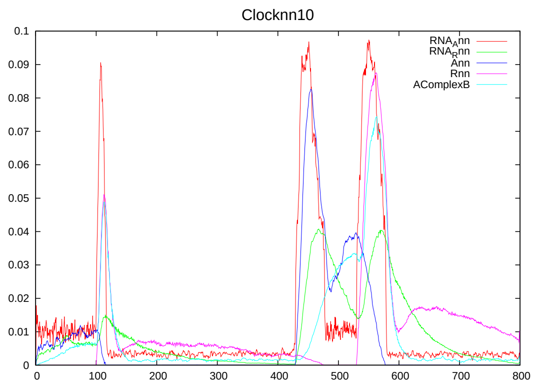


(f) ODE solution for the system scaled by 100

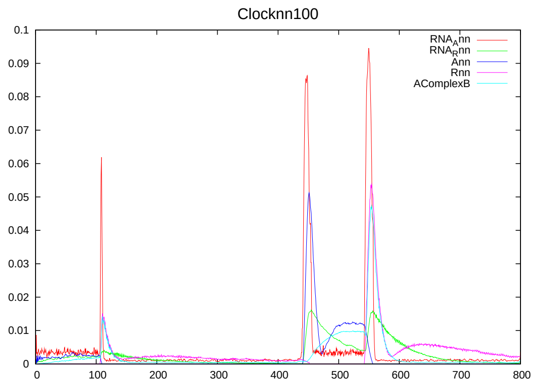
Figure 5.3: The effect of scaling on the simulation of the circadian clock model. The average of 20 different runs of the simulation is compared to the solution of the corresponding ODEs for the original model and the original model scaled by factors of 10 and 100 respectively.



(a) original model



(b) scaled by 10



(c) scaled by 100

Figure 5.4: Plots of the changing variance for the circadian clock model. For each scale, the standard deviation scaled by the maximal mean is plotted. We can see that the variance decreases as the scale increases.

Example 17. The last example we look at is where we fail to observe any convergence behaviour. Consider a simple *repressilator* model from [1]. The model consists of three *gene gates* (processes $Gene\langle a, b\rangle$, $Gene\langle b, c\rangle$ and $Gene\langle c, a\rangle$), each capable of producing a different protein ($Protein\langle b\rangle$, $Protein\langle c\rangle$ and $Protein\langle a\rangle$ respectively) if in an active state. A gene gate can be *blocked* to a blocked state where it is unable to produce the protein. This is triggered by an abundance of a protein from another gene gate – $Protein\langle a\rangle$ blocks $Gene\langle a, b\rangle$, $Protein\langle b\rangle$ blocks $Gene\langle b, c\rangle$ and $Protein\langle c\rangle$ blocks $Gene\langle c, a\rangle$. The proteins can also degrade at a certain rate. The following system defines this in $\mathcal{S}\pi$:

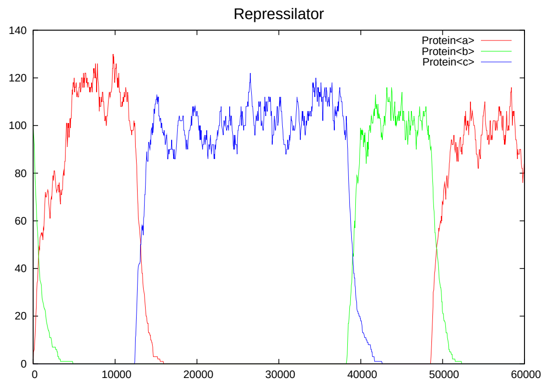
$$\begin{aligned} Gene(a, b) &\overset{\text{def}}{=} \tau@t.(Protein\langle b\rangle | Gene\langle a, b\rangle + ?a.\tau@u.Gene\langle a, b\rangle), \\ Protein(b) &\overset{\text{def}}{=} !b.Protein\langle b\rangle + \tau@d \end{aligned}$$

where the top-level process is

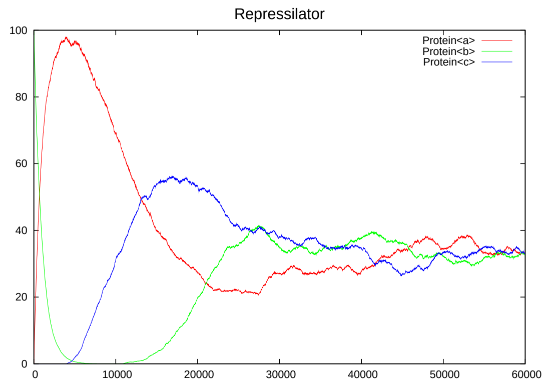
$$Repressilator = Gene\langle a, b\rangle | Gene\langle b, c\rangle | Gene\langle c, a\rangle$$

Simulating this system, we can observe alternate cycles of protein production, characterised by an abundant protein and two blocked gene gates. One of the blocked gates is repressed by the produced protein, whereas the other is not. If the repressed gate unblocks, it is likely to get blocked, because of the presence of the blocking protein. If the other gate unblocks, it starts producing a protein that represses the currently active gate – the cycle changes. The order of the different periods as well as their lengths is highly stochastic – see [1] for a detailed investigation, also concerning the influence of changing rates of the different channels. We add that the situation does not change with scaling – in presence of larger populations of genes, they tend to get blocked and unblocked in the same cycles, as the proportion of produced proteins also changes.

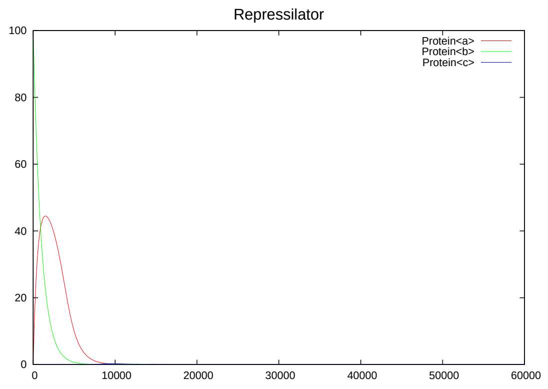
This behaviour is clearly discrete, with the genes and blocked genes being in populations of only 0 and 1. Also, since the system starts in an "equilibrium" where each gene produces proteins blocking the other genes, there is no reason for oscillation in the deterministic continuous case. Therefore in order to examine the continuous semantics, we artificially block two of the genes. However, this doesn't change much, as the system tends to the "equilibrium" state after the first artificial oscillation ends. See Figure 5.5 for the plots of the simulation and ODE solution for the original system and Figure 5.6 for the effects of scaling.



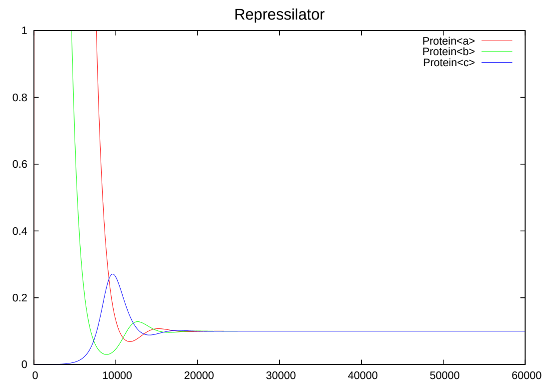
(a) simulation



(b) replicated simulation

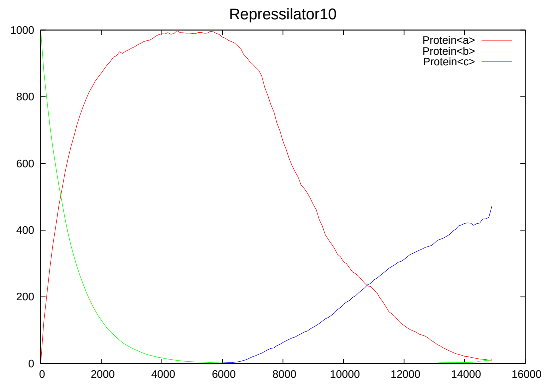


(c) ODE solution

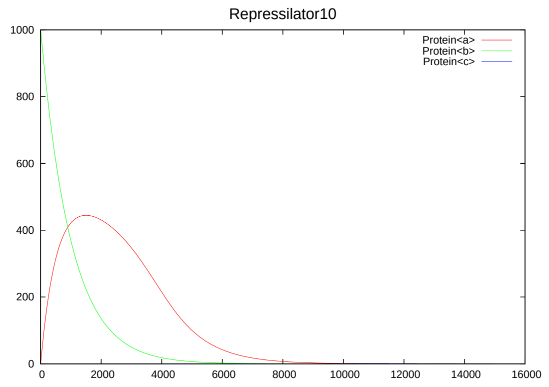


(d) detail of ODE solution

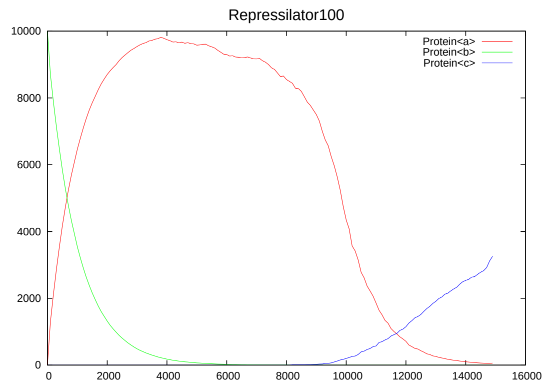
Figure 5.5: Sample simulation and ODE solution of the repressilator model. The ODE solution tends to go to the equilibrium state.



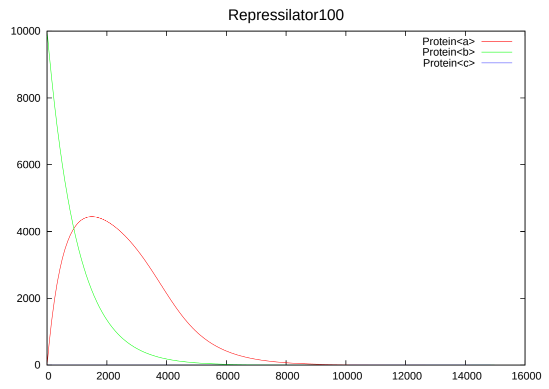
(a) Replicated simulation for the system scaled by 10



(b) ODE solution for the system scaled by 10



(c) Replicated simulation for the system scaled by 100



(d) ODE solution for the system scaled by 100

Figure 5.6: The effect of scaling on the simulation of the circadian clock model. The average of 20 different runs of the simulation is compared to the solution of the corresponding ODEs for the original model and the original model scaled by factors of 10 and 100 respectively. This suggests that the simulations do not converge to the ODE solution.

6 Spatial extension of stochastic π calculus

In this chapter we define a simple spatial extension of $\mathcal{S}\pi$, which we call $\mathcal{L}\pi$. As we argued in the introduction, our main aim is to provide an extension that will allow the reuse of existing $\mathcal{S}\pi$ models as well as keep the alternative continuous semantics. We will also only consider the case of static compartments, as has been done in [11].

We first introduce the main ideas and give an informal justification why they might be a suitable representation of the physical nature of the problem. After that, we give a formal definition of $\mathcal{L}\pi$, defining the syntax and operational semantics. This will also require slight changes to the original semantics of $\mathcal{S}\pi$, caused by the presence of explicit volume in $\mathcal{L}\pi$ – we describe these changes. We show how the aggregation result translates to $\mathcal{L}\pi$ and give an efficient simulation algorithm. We support the formalism by presenting two examples – one a simple adaptation of an epidemic SIR model, another an original example using the location structures to model virus spreading in plant tissue. As the next step, we extend the continuous semantics of $\mathcal{S}\pi$ to $\mathcal{L}\pi$ and give a simple example. We conclude the chapter by looking at the expressive power of $\mathcal{L}\pi$ and show that it can be "emulated" by $\mathcal{S}\pi$.

We go back to an argument before $\mathcal{S}\pi$ and generalize the example from Section 2.1.5. Consider two containers, say C_1 and C_2, with constant volumes V_1 and V_2 respectively. Assume that C_1 and C_2 are connected in some way, say by sharing a part of their surface of area A, through which molecules of the species X can "cross", after approaching it at a distance m (say this expresses some kind of permeability of the boundary). Let C_1 contain molecules of X (with radius r), uniformly distributed. We can take the average speed v of X molecules with respect to a point P on the boundary and consider the volume a single molecule sweeps in time δt, $\delta V_{\text{cross}} = \pi d^2 v \delta t$ where $d = m + r$. See Figure 6.1.

Then we have the probability of X hitting P to be $c \cdot \delta t$, where c is a constant specific for X and V_2. If we now have N X molecules, the probability of any molecule hitting *any* point on the surface will be proportional to $N \cdot A \cdot c\delta t$. Looking at this as a hazard rate of the distribution until the next movement of an X molecule from C_1 to C_2, we get this to be exponentially distributed with parameter depending on the species X and the two compartments C_1 and C_2. We can treat reactions happening inside C_1 and C_2 independently and in the same fashion as in the argument in the section 2.1.5. Last but not least, we ignore the possibility of reactions occurring "between" two compartments. Therefore our system still yields a CTMC.

We can now take a *system* of compartments (possibly nested, but not overlapping), assuming that each pair can share some surface and thus allow movement of molecules. We can abstract from the area of this surface and the permeability constant and just consider a single number, which we will call the *movement constant* for the two compartments. Clearly if the two compartments are not directly connected, this number is zero. We also abstract from the fact that the compartments are actual physical containers – they can represent regions of different properties inside another

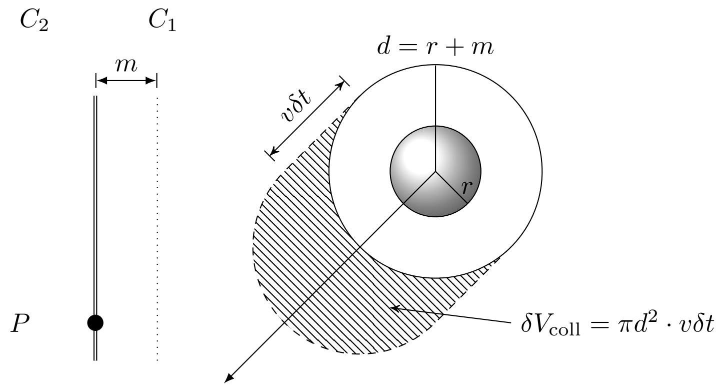


Figure 6.1: The X molecule (represented by the sphere of radius r) moves towards a point P on the boundary between the compartments, sweeping volume δV_{coll} in a short time interval δt.

compartment. We will therefore use the word *location* to denote them. This system of locations can be represented as a graph with directed and weighted edges, with the weight being the movement constant. The nodes can also carry weight corresponding to the volume of the respective locations.

As our plan is to re-use existing $\mathcal{S}\pi$ models, the direction of our abstraction is clear at this point – we assume each of the nodes of the above graph (we will call such graphs *location graphs*) can contain a $\mathcal{S}\pi$ process. The behaviour within locations will be the same as given by $\mathcal{S}\pi$, with the difference that the volume is no longer 1. Additionally, the processes (the prime processes in parallel composition) in the locations will be allowed to "move" between the locations of the graph, with rates given by a function representing the movement constants.

6.1 Syntax

We will extend the syntax of $\mathcal{S}\pi$ by adding a layer of graphs. The nodes of these will have explicit names, and so

$\mathcal{N}_L$

Definition. Define the set of *location names* $\mathcal{N}_L$. We will use $l, m, n, \dots$ to denote these.

The location graphs are a finite set of locations, each containing an ordinary $\mathcal{S}\pi$ process, together with a function expressing the *volume* of each location and a *movement function* expressing the rate of movement between locations for each prime process.

$\mathcal{L}$
$[\dots, l_i\colon P_1, \dots]_{v,m}$

Definition. Define the set of *location graphs* $\mathcal{L}$ to be the minimal set such that

$$[l_1\colon P_1, \dots, l_n\colon P_n]_{v,m} \in \mathcal{L}$$

where $L = \{l_1, \dots, l_n\} \subseteq \mathcal{N}_L$ is a set of location names, $P_i \in \mathcal{P}$ for all $i = 1, \dots n$, $v\colon L \to \mathbb{R}$ is a *volume function* and $m\colon \widehat{\mathcal{P}} \times L \times L \to \mathbb{R}$ is a *movement function.*

If $G = [l_1\colon P_1, \dots, l_n\colon P_n]_{v,m}$ we can write that $l_i \in G$ and $l_i\colon P_i \in G$ for each $i = 1, \dots, n$.

(G, E)

A *system of* $\mathcal{L}\pi$ is a tuple consisting of a graph G and an evironment E such that for all $l_i\colon P_i \in G$, P_i is valid with respect to E.

6.2 Semantics

The semantics of $\mathcal{L}\pi$ is a direct extension of that of $\mathcal{S}\pi$, justified by the argument at the beginning of this chapter. The evolution inside locations is directed by the internal $\mathcal{S}\pi$ processes. The only modification is in the resulting rates, which are influenced by the volume function – the rate of *communication between two processes* should be inversely proportional to the volume. At this place, we are forced to slightly modify the semantics of $\mathcal{S}\pi$. The problem is that in its original form, it is not possible to distinguish between a transition resulting from communication and from a silent action – a required property since the rates of silent actions should not depend on the volume of the enclosing compartment. For this purpose, we define a new kind of transition τ_a where $a \in \mathcal{N}$, which will be treated in the same way as $\tau@r_a$, but will be distinguished when considering the semantics of transitions inside locations. The transition τ_a will arise when two processes communicate on the channel a (that is in the rules Com and Close which we need to modify). We also need to pay attention to complexes – the components of these should be considered close in the compartment and therefore the rate of their communication should not depend on the volume. This can be treated by modifying the rule Res – if the inside of a restriction on a channel e communicates on a channel that is not private (so does a transition τ_a, $a\,\mathrm{new}\,e$), the resulting transition should be influenced by the volume and so is τ_a. Otherwise, it is an internal action of the complex, not influenced by the volume of the compartment and so is $\tau@r_a$. For this to make sense, we have to assume that subprocesses of a (prime) complex can communicate with each other only on the private channel; however, the semantics will be well defined even if this is not the case.

Apart from transitions inside locations, the only other transitions are those corresponding to processes moving from one location to another. The rate of this is given by the movement function. A prime process can move from one location to another if it is inside parallel composition in the location. At this point we make a design decision and allow movement of only summations and identifier instances – for movements of restrictions we would have to make sure that the moving process is a prime process, which would complicate the semantics in a significant way. We can justify this by the fact that complexes are "too complicated" to move. We leave it for future work to find models which would benefit from movements of complexes.

The movement will be represented by a *lifting action*, removing a process from a parallel composition.

Definition. Define the *lifting action* to be $\mathrm{lift}(P)$, where $P \in \mathcal{P}$. Define the set of all actions of $\mathcal{L}\pi$ to be $\mathcal{A}_{\mathcal{L}}$, including all the actions of $\mathcal{S}\pi$, $\mathcal{A}_+$ and all the lifting actions. $\mathrm{lift}(P)$ $\mathcal{A}_{\mathcal{L}}$

Definition (transition semantics)**.** Define a *pre-transition relation* to be a multi relation $\cdot \overset{\cdot}{\longrightarrow} \cdot \Subset \mathcal{L} \times \mathcal{A}_\tau \times \mathcal{L}$ defined as the restriction of the multi relation $\cdot \overset{\cdot}{\longrightarrow} \cdot \Subset \mathcal{L} \times \mathcal{A}_{\mathcal{L}} \times \mathcal{L}$ inductively defined by the rules in the Figure 6.2. $\cdot \overset{\cdot}{\longrightarrow} \cdot$

The rule Inside concerns the transitions inside the locations, that involve communication on "global" channels. The rule InsideTau concerns local transitions which do not depend on the volume. The rule rule LiftSum is the base case for lifting summations and the rule LiftID for identifiers defining summations. The rule LiftPar allows lifting through trees of parallel compositions. The rule LiftNew allows lifting from restrictions only when the restricted name isn't in the lifted process. Finally, the rule Move defines movement, where it puts a lifted process from one location to another (into the topmost parallel composition), with the rate determined by the movement function (which has to be non-zero).

6.3 Simulation

We can use structural congruence of the $\mathcal{S}\pi$ processes inside locations to provide a theorem analogous to Theorem 3.3.

Theorem 6.1. If $G = [l_1 : P_1, \dots, l_n : P_m]$ and P_i has a standard form $(\mathrm{new}\,\varphi_i)\{\!|n_{i,1}{\times}P_{i,1}, \dots, n_{i,m_i}{\times}P_{i,m_i}|\!\} = (\mathrm{new}\,\varphi_i)P_i'$ then

$$\begin{aligned}
\mathit{Trans}(G) = {} & \{\!|G \xrightarrow{\tau@r_a/v(l_i)} [\cdots l_i : Q_i, \dots]_{v,m} : P_i \xrightarrow{\tau_a} Q_i|\!\} \uplus \\
& \{\!|G \xrightarrow{\tau@r} [\cdots l_i : Q_i, \dots]_{v,m} : P_i \xrightarrow{\tau@r} Q_i|\!\} \uplus \\
& \{\!|G \xrightarrow{\tau@m(P_{i,k}, l_i, l_j)} [\dots, l_i : (\mathrm{new}\,\varphi_i)(P_i' \setminus P_{i,k}), \dots, l_j : P_{i,k}|P_j, \dots]_{v,m} \\
& \quad \mathrm{fn}(P_{i,k}) \cap \varphi_i = \emptyset,\ m(P_{i,k}, l_i, l_j) \neq 0|\!\}.
\end{aligned}$$

Proof. This follows from the Theorem 3.3 and a similar argument concerning the movement transitions. ■

$$\textsc{Inside}: \quad \frac{P \xrightarrow{\tau_a} Q}{[\ldots, l_i\colon P, \ldots]_{v,m} \xrightarrow{\tau@r_a/v(l_i)} [\ldots, l_i\colon Q, \ldots]_{v,m}}$$

$$\textsc{InsideTau}: \quad \frac{P \xrightarrow{\tau@r} Q}{[\ldots, l_i\colon P, \ldots]_{v,m} \xrightarrow{\tau@r} [\ldots, l_i\colon Q, \ldots]_{v,m}}$$

$$\textsc{LiftSum}: \quad \frac{}{\sum_{i\in I}\alpha_i.P_i \xrightarrow{\mathrm{lift}(\sum_{i\in I}\alpha_i.P_i)} \mathbf{0}}$$

$$\textsc{LiftID}: \quad \frac{}{A\langle\widetilde{\psi}\rangle \xrightarrow{\mathrm{lift}(A\langle\widetilde{\psi}\rangle)} \mathbf{0}} \ \text{if } A(\widetilde{x}) \overset{\mathrm{def}}{=} \sum_{i\in I}\alpha_i.P_i$$

$$\textsc{LiftPar}: \quad \frac{P_1 \xrightarrow{\mathrm{lift}(Q)} P_1'}{(P_1|P_2) \xrightarrow{\mathrm{lift}(Q)} (P_1'|P_2)}$$

$$\textsc{LiftNew}: \quad \frac{P \xrightarrow{\mathrm{lift}(Q)} P'}{(\mathrm{new}\, x@r)P \xrightarrow{\mathrm{lift}(Q)} (\mathrm{new}\, x@r)P'}\ , \ x \notin \mathrm{fn}(Q)$$

$$\textsc{Move}: \quad \frac{P \xrightarrow{\mathrm{lift}(Q)} P' \qquad m(Q, l_i, l_j) \neq 0}{[\ldots, l_i\colon P, \ldots, l_j\colon R, \ldots]_{v,m} \xrightarrow{\tau@m(Q,l_i,l_j)} [\ldots, l_i\colon P', \ldots, l_j\colon (Q|R), \ldots]_{v,m}}$$

Figure 6.2: The transition rules for the location graphs. The ... mean that the locations not listed remain unchanged between left and right components of members of $\rightarrow$.

The above theorem gives a straightforward modification of the Gillespie algorithm for $\mathcal{L}\pi$. An important observation is that we can use the same techniques from the $\mathcal{S}\pi$ version of this algorithm to obtain the internal transitions inside locations – this will become useful in the implementation. See Algorithm 5.

We end this section with two interesting examples illustrating the flexibility of $\mathcal{L}\pi$.

Example 18. We look at a simple extension of a model in $\mathcal{S}\pi$.

Consider the environment E_{SIR} with the defining equations

$$\begin{aligned} S &\overset{\mathrm{def}}{=} ?\mathit{infect}.I, \\ I &\overset{\mathrm{def}}{=} !\mathit{infect}.I + \tau@r_{\mathrm{recover}}.R, \\ R &\overset{\mathrm{def}}{=} \mathbf{0} \end{aligned}$$

and the top-level process

$$\mathit{System} = 100 \times S|10 \times I.$$

This is a simple epidemics model, such as the one due to Kermack-McKendrick [40]. In this model, there is a population of individual *susceptible* to a disease, transfered from *infected* individuals, who *recover* after a period of time.

One can be interested in the dynamics of this model in the case there is an active quarantining, that is if the infected individuals are likely to get diagnosed and isolated until they recover. We can model this as a location graph with two nodes – one the original "world" (node a), the other the quarantine (node b). The infected will then be "allowed" to move from a to b (the rate of the movement, say r_{diagnose}, can represent the rate of how fast they become diagnosed) and the

Algorithm 5 The Gillespie algorithm for $\mathcal{L}\pi$.

```
1: Start with a graph $G = [l_1 : P_1, \ldots, l_n : P_n]_{v,m}$ where $P_i$'s have standard form as in Theorem 6.1
2: repeat
3:   for all $i = 1, \ldots, n$ do
4:     collect $T_i' = \{\!|P_i \xrightarrow{\tau@r} Q_i|\!\}$ as in Algorithm 4
5:     set $T_i = \{\!|G \xrightarrow{\tau@r} [\ldots, l_i : Q_i, \ldots]_{v,m}|\!\}$
6:     collect $T_i'' = \{\!|P_i \xrightarrow{\tau_a} Q_i|\!\}$ as in Algorithm 4
7:     add $\{\!|G \xrightarrow{\tau@r_a/v(l_i)} [\ldots, l_i : Q_i, \ldots]_{v,m}|\!\}$ to $T_i$
8:   end for
9:   for all $i, j, k$ such that $m(i, j, P_{i,k}) > 0$ and $\mathrm{fn}(P_{i,k}) \cap \varphi_i = \emptyset$ do {collect all movement transitions $T_m$}
10:    insert $n_{i,k}$ copies of $G \xrightarrow{\tau@m(i,j,P_{i,k})} [\ldots, l_i : (\mathrm{new}\, \varphi_i)(P_i' \setminus P_{i,k}), \ldots, l_j : P_{i,k}|P_j, \ldots]_{v,m}$ into $T_m$
11:  end for
12:  let $T = T_1 \uplus \cdots \uplus T_n \uplus T_m$
13:  let $r_{\mathrm{total}} = \sum_{S \xrightarrow{\tau@r} U} r$
14:  randomly select a transition $G \xrightarrow{\tau@r} G' \in T$ with probability $r/r_{\mathrm{total}}$
15:  generate $\delta t$ from $\mathrm{Exp}(r_{\mathrm{total}})$
16:  set $t = t + \delta t$, set $G = G'$
17: until $t = t_{\mathrm{stop}}$
```

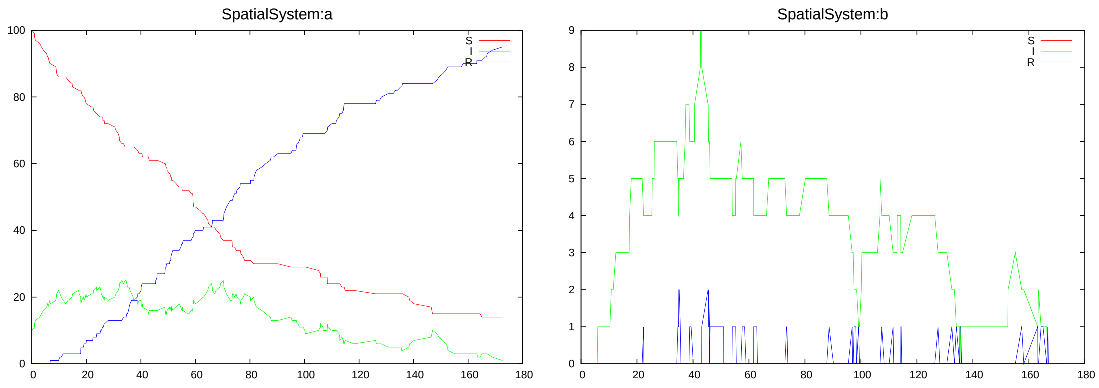
recovered will be allowed to move from b to a (the rate of movement, say r_{observe}, can represent the "observation" period to determine an individual has recovered). The graph then would be

$$[a : \mathit{System}, b : \mathbf{0}]_{v,m}$$

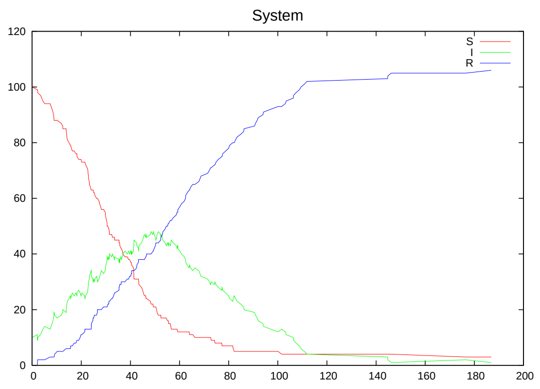
where $m(a, b, I) = r_{\mathrm{diagnose}}$, $m(b, a, R) = r_{\mathrm{discharge}}$ and 0 otherwise and $v(a) = v(b) = 1$. See Figure 6.3 for a sample simulation and comparison with the single compartmental model. One quantity of interest can be the proportion of susceptible population that has not become infected during the spread of the disease and the effect of quarantining on this number.

Example 19. This example will illustrate a slightly different approach to the above. Consider a hypothetical plant tissue. This tissue consists of cells arranged in a two dimensional grid. A cell can be attacked by a virus. A hypothesis is that in this case, it sends out a signal to the neighbouring cells, which in turn become more resistant to the virus and thus eventually prevent its spreading to the whole tissue. We will show how $\mathcal{L}\pi$ could be used to model this situation and so to carry out experiments in-silico to confirm the hypothesis.

Our location graph will represent the structure of the tissue – we take a grid with only adjacent nodes connected. Each location will correspond to a cell – initially, it will contain a *Cell* process. The *Virus* process will be able of attacking the cell. In that case, the cell releases warnings to the neighbouring cells – it will create several *Warning* processes, that are allowed to move to the neighbours – and switches to the mode fighting the virus. The life of the cell will be represented by the process *Life* and the resistance against the virus by the *Resistance* processes. The resistance processes will be able to attack the virus (output action on the channel *defeat*), while the virus attacks the life of the cell (output action on the channel *fight*) – the likelihood of cell surviving therefore depends on the number of resistance cells it releases. When a virus wins, the cell gets defeated and the virus multiplies, otherwise a resistance process destroys the virus and notifies the cell (output action on the channel *defeated*) which then switches back to the normal state (but with resistance to the virus). When a cell gets warned (communicates with a warning process), it switches to the resistant state (the process *RCell*), which is identical to the *Cell* with the difference that it releases more *Resistance* processes.



(a) System with quarantine



(b) Original system

Figure 6.3: Simulation of the SIR model with active quarantining and the original model with single location. We can be interested in comparing the number of susceptible individuals who do not become infected (around 20 in case of quarantining and 5 in the original model).

Therefore take the environment

$$\begin{aligned}
\mathit{Cell} &= ?\mathit{attack}.(\mathit{Life}|6\times \mathit{Reistance}|4\times \mathit{Warning})+?\mathit{warn}.\mathit{RCell},\\
\mathit{RCell} &= ?\mathit{attack}.(\mathit{Life}|20\times \mathit{Reistance}|4\times \mathit{Warning})+?\mathit{warn}.\mathit{RCell},\\
\mathit{Resistance} &= !\mathit{defeat}.!\mathit{defeated} + \mathit{delay}@\mathit{expire},\\
\mathit{Life} &= ?\mathit{fight}+?\mathit{defeated}.\mathit{RCell},\\
\mathit{Virus} &= !\mathit{attack}.(!\mathit{fight}.(2\times \mathit{Virus})+?\mathit{defeat})
\end{aligned}$$

and the graph

$$\mathit{Tissue} = [c_{x,y}\colon \mathit{Cell}|\text{if }(x,y)=(1,1)\text{ then } \mathit{Virus} : x,y=0,\dots,3]_{v,m}$$

with $v(c_{x,y}) = 1$ and

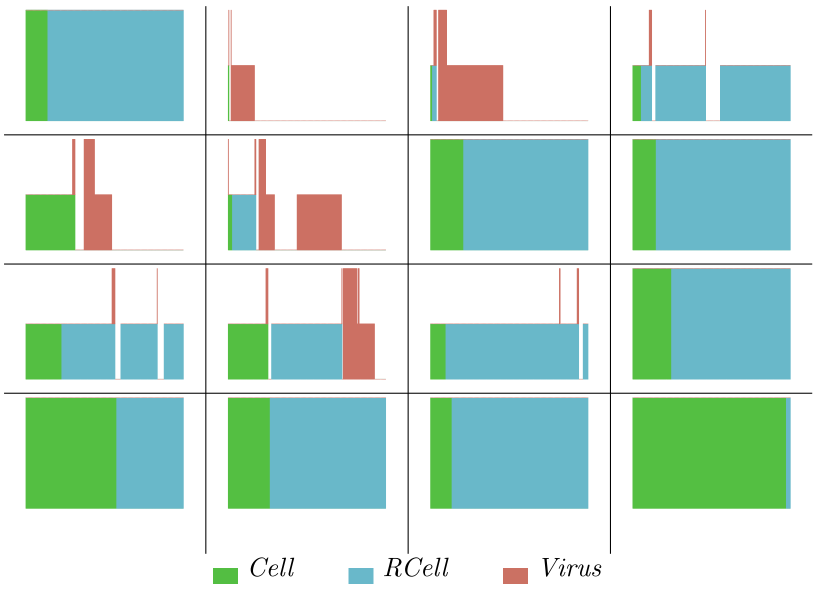
$$m(c_{x,y}, c_{x+\Delta x,y+\Delta y}) = \begin{cases} \mathit{move} & \text{if } \Delta x, \Delta y \in \{-1,0,1\} \text{ and } |\Delta x|+|\Delta y| = 1,\\ 0 & \text{otherwise.}\end{cases}$$

The picture on Figure 6.4 should make the above more clear.

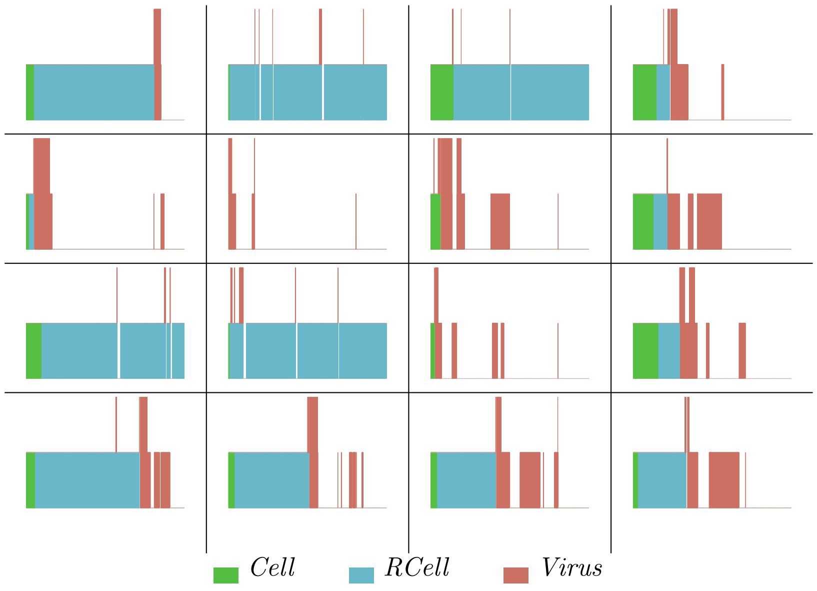
We can execute a simulation of this model to see the effects of different parameters of the model, such as the numbers of *Resistance* processes released by the cell or rates of individual channels. See Figure 6.5 for two sample simulation traces.

$$\begin{tikzcd}
c_{0,0}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{0,1}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{0,2}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{0,3}: \mathit{Cell} \arrow[d, leftrightarrow] \\
c_{1,0}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{1,1}: \mathit{Cell} | \mathit{Virus} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{1,2}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{1,3}: \mathit{Cell} \arrow[d, leftrightarrow] \\
c_{2,0}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{2,1}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{2,2}: \mathit{Cell} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & c_{2,3}: \mathit{Cell} \arrow[d, leftrightarrow] \\
c_{3,0}: \mathit{Cell} \arrow[r, leftrightarrow] & c_{3,1}: \mathit{Cell} \arrow[r, leftrightarrow] & c_{3,2}: \mathit{Cell} \arrow[r, leftrightarrow] & c_{3,3}: \mathit{Cell}
\end{tikzcd}$$

Figure 6.4: Location graph of the plant tissue model. The locations are arranged in a grid, where only adjacent locations are connected. Each location contains a *Cell* process and $c_{1,1}$ additionally contains a *Virus* process. The *Virus* is allowed to move between locations. When it attacks a cell, the cell releases *Warning* processes, which are allowed to spread to neighbouring locations to make their cells more resistant. Nothing else is allowed to move.



(a) Virus contained



(b) Virus spreading

Figure 6.5: Sample simulation of the plant patogen model. Each cell in the grid represents time evolution of the corresponding compartment. The system starts with the virus in the location $c_{1,1}$. Figure (a) shows an example of a simulation where the virus is contained after attacking the neighbouring cells of $c_{1,1}$. Figure (b) shows a simulation where the virus spreads to the neighbouring cells and survives.

6.4 Continuous semantics

In this section we present the continuous semantics for $\mathcal{L}\pi$. The argument for its suitability is the same as for the case of $\mathcal{S}\pi$, using the Theorem 6.1.

We need to extend the definition of prime processes. Because the processes are allowed to move, we cannot consider prime processes for the locations individually but need to take the whole graph into account.

Definition. Let (G, E) be an $\mathcal{L}\pi$ environment. The *prime processes* of (G, E) is the set $\widehat{\mathcal{P}}(G, E)$ of all $\mathcal{S}\pi$ prime processes P such that $G \longrightarrow^* G'$ and $G' \equiv [\dots, l_i\colon P|Q, \dots]_{v,m}$.

Now each location will contain populations of prime processes – we need separate quantity functions for each location–prime process combination.

Definition. For each $P \in \widehat{\mathcal{P}}(G, E)$ and a location $l \in G$, define the *quantity function of* P to be $[P]_l\colon \mathbb{R} \to \mathbb{R}$. $[P]_l$

Definition. The *system of differential equations of an initial graph* $G = [l_1\colon P_1, \dots, l_n\colon P_n]_{v,m}$ *with respect to an environment* E consists of the following, for each $P \in \widehat{\mathcal{P}}(G, E)$ and location $l \in G$, $\frac{\mathrm{d}[P]_l}{\mathrm{d}t}$

$$\begin{aligned}\frac{\mathrm{d}[P]_l(t)}{\mathrm{d}t} = &\sum_{(a,P_1,P_2)\in\mathrm{Enter}_{ch,G,E}(P)} r_a \cdot c([P_1]_l(t), [P_2]_l(t))/v(l) + \sum_{(r,Q)\in\mathrm{Enter}_{\tau,G,E}(P)} r\cdot[Q]_l \\ &- \sum_{(a,Q)\in\mathrm{Exit}_{ch,G,E}(P)} r_a\cdot c([P]_l(t), [Q]_l(t))/v(l) - \sum_{r\in\mathrm{Exit}_{\tau,G,E}(P)} r\cdot [P]_l(t) \\ &+ \sum_{m(k,l,P)\neq 0} m(k,l,P)\cdot[P]_k(t) - \sum_{m(l,k,P)\neq 0} m(l,k,P)\cdot[P]_l(t)\end{aligned}$$

where the Enter and Exit sets are defined as in $\mathcal{S}\pi$ with respect to $\widehat{\mathcal{P}}(G, E)$ and the function $c(\cdot,\cdot)$ defined in the same way as for $\mathcal{S}\pi$(see Chapter 4).

Definition. The *continuous semantics* of a graph G with respect to an environment E is

$$\{[P]_l : P \in \widehat{\mathcal{P}}(G, E),\ l \in G\}$$

where $[P]_l$ satisfy the initial conditions given by

$$[P]_l(0) = P_l\#P$$

where $G = [\dots l\colon P_l, \dots]_{v,m}$.

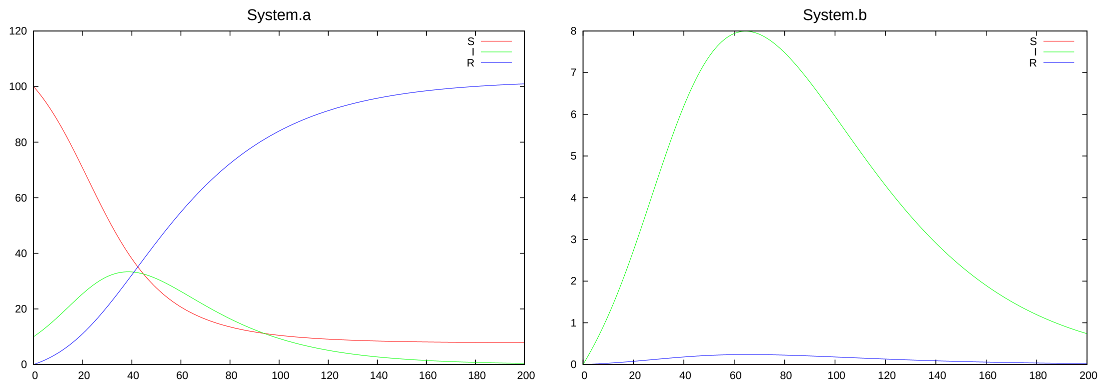
Example 20. Consider the SIR model with quarantine. We get the following Enter and Exit multisets (only listing the non-empty ones)

$$\begin{aligned}\mathrm{Exit}_{ch,G,E}(S) &= \{\!|(\mathit{infect}, I)|\!\},\\ \mathrm{Enter}_{ch,G,E}(I) &= \{\!|2\times(\mathit{infect}, S, I)|\!\},\\ \mathrm{Exit}_{ch,G,E}(I) &= \{\!|(\mathit{infect}, S)|\!\},\\ \mathrm{Exit}_{\tau,G,E}(I) &= \{\!|(r_{\mathrm{recover}})|\!\},\\ \mathrm{Enter}_{\tau,G,E}(R) &= \{\!|(r_{\mathrm{recover}}, I)|\!\}.\end{aligned}$$

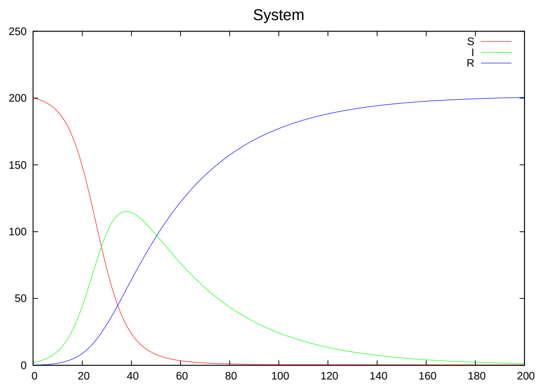
The system of ODEs of (G, E) is

$$\begin{aligned}
\frac{\mathrm{d}[S]_a(t)}{\mathrm{d}t} &= -r_{infect} \cdot [S]_a(t) \cdot [I]_a(t), \\
\frac{\mathrm{d}[I]_a(t)}{\mathrm{d}t} &= r_{infect} \cdot [S]_a(t) \cdot [I]_a(t) - r_{\mathrm{recover}} \cdot [I]_a(t) - r_{\mathrm{diagnose}} \cdot [I]_a(t), \\
\frac{\mathrm{d}[I]_b(t)}{\mathrm{d}t} &= r_{\mathrm{diagnose}} \cdot [I]_a(t) - r_{\mathrm{recover}} \cdot [I]_b(t), \\
\frac{\mathrm{d}[R]_a(t)}{\mathrm{d}t} &= r_{\mathrm{recover}} \cdot [I]_a(t) + r_{\mathrm{discharge}} \cdot [R]_b, \\
\frac{\mathrm{d}[R]_b(t)}{\mathrm{d}t} &= r_{\mathrm{recover}} \cdot [I]_b(t) - r_{\mathrm{discharge}} \cdot [R]_b(t).
\end{aligned}$$

See Figure 6.6 for a numerical solution to this system of ODEs and a comparison with the solution for the original model.



(a) System with quarantine



(b) Original system

Figure 6.6: Numerical solutions to the ODEs from the SIR model with quarantining compared with the results for the original model.

6.5 Relationship to $\mathcal{S}\pi$

We look at the expressive power of $\mathcal{L}\pi$. We give an informal argument (which can be easily formalized) justifying that any $\mathcal{L}\pi$ system can be emulated by a $\mathcal{S}\pi$ system. Suppose we are given a location graph with n locations. We need to make sure that

(1) the prime processes exist in n disjoint groups, able to react only with the processes from the same group (with reaction rates modified by different volumes assigned to each group) and

(2) that summations (or identifiers defining summations) can "move" between these groups.

To guarantee (1), note that reactions between processes can happen on "global" channels – we can introduce a new version of each channel for each location, with the rate modified by the volume of the location. For each defining equation in the $\mathcal{S}\pi$ environment, we can introduce n new defining equations, each using one of the versions of each channel. Note that this does not influence restricted channels – if two processes can create a complex, the sending process has to send the restricted name through a channel – this will make sure they are in the same location.

To guarantee (2), we have to make sure every summation can "move" to a summation corresponding to a different location. Without loss of generality we can assume that each summation is defined by an identifier in the environment. Then for each non-zero movement constant, this summation can contain a silent transition with the movement constant as the rate, evolving into a version of the identifier for the target location.

The situation would change if complexes are allowed to move between locations – we would have to make sure all their subprocesses "move" to the new location at the same time. In case there is more than two, this would require a synchronization primitive not present in $\mathcal{S}\pi$.

We only add that despite the same expressive power, $\mathcal{L}\pi$ certainly makes it more convenient to define models involving static compartments, as can be seen from the examples.

6.6 Summary

We introduced a spatial extension of $\mathcal{S}\pi$ for static compartments, called $\mathcal{L}\pi$. We gave its syntax and semantics, in a style similar to $\mathcal{S}\pi$ and also extended the continuous semantics to $\mathcal{L}\pi$. We presented two examples justifying the use of $\mathcal{L}\pi$. Finally, we argued that although the expressive power of $\mathcal{L}\pi$ and $\mathcal{S}\pi$ are the same, $\mathcal{L}\pi$ provides more convenient modelling framework.

7 Implementation

In this chapter we present an implementation of some of the theoretical concepts described earlier. The initial aim for the implementation was to produce a convenient tool to explore (experimentally) the relationship between the discrete and continuous semantics of the stochastic π calculus. Another aim was to provide an extensible platform on which further extensions, such as the outlined spatial calculus, can be built. The tool was mainly inspired by the *SPiM* abstract machine [32] and should serve as an alternative when SPiM is not fully available (e.g. under Linux operating system, no graphical interface is provided) and comparison with the continuous semantics of models in CGF is desired. Moreover, written in the Java programming language, it provides an extensible $\mathcal{S}\pi$ calculus implementation to a wider range of developers.

Briefly, the developed tool, named JSPiM is an interpreter for $\mathcal{S}\pi$ and $\mathcal{L}\pi$. Similar to SPiM, it consists of a simple text editor for writing models. The models are described in a syntax similar to that of the formal definition of $\mathcal{S}\pi$ and $\mathcal{L}\pi$. The editor, as opposed to SPiM, provides syntax error reporting, making it more convenient to write new models. In addition to description of the environment and top-level processes, *commands* can be specified. These include simulation of the top-level $\mathcal{S}\pi$ process (both single traces and mean and variance from replicated runs) ODE solution of the continuous semantics if the system is in CGF and a simulation of location graphs of $\mathcal{L}\pi$. JSPiM presents the results of the commands visually (using the chart library *JFreeChart*). Moreover, it is able to export the resulting data together with *GNUPlot* description files that can be used to automatically produce diagrams suitable for papers (most of the graphs in this report were generated in this way). For long running executions, JSPiM supports non-interactive mode which can be run without a graphical interface.

We first describe the high level architecture overview of JSPiM. Intentionally, we only include the $\mathcal{S}\pi$ part of JSPiM and describe the $\mathcal{L}\pi$ extension later, mirroring the order of the theoretical development. We list the main package structure in which the code is organized and describe the used external libraries. After this, we describe how JSPiM represents the different structures from $\mathcal{S}\pi$. We start with a description of how the syntax translates (via grammars written in the *ANTLR* compiler generator) to the internal representation of processes. We describe this representation, which closely follows the formal description from Chapter 3. JSPiM uses higher level collections to allow efficient implementation of the simulation algorithm and ODE generations – we describe these and show how they relate to the formal description, suggesting a proof of correctness of our implementation. We give an overview of the commands JSPiM provides and show how they use the collections.

The implementation of $\mathcal{L}\pi$ demonstrates the extensibility of JSPiM. We describe how the implementation of $\mathcal{L}\pi$ structures fits into the core architecture and how they relate to the formal development in Chapter 6.

We conclude the chapter with remarks on how testing was carried out during development of JSPiM and also briefly and informally evaluate how its performance compares to SPiM, using models from the model library in the Appendix B.

7.1 Architecture overview

JSPiM is written in the Java 1.5 programming language. We tried to follow the best practices of software engineering, to provide a solid implementation as well as a platform for further extensions. The high level package structure of JSPiM is the following:

- The whole code falls under the package jspim. This contains the main class JSPIM that handles different command line options and sets up the user interface or non-interactive execution.
- The user interface components are in the gui package. These include the main window GUI as well as a simple text editor EditorPane and chart windows for displaying the results.
- Package syntax consists of the auto-generated code from *ANTLR* parser generator and provides classes for translation of stream of characters into the representation of stochastic π calculus.
- The representation of processes and environments is in the package processes.
- Package collections defines higher level structures that provide primitives for stochastic simulation and ODE generation, such as the classes ProcessesCollection and ReactingCollection for efficient simulation and ODECollection for ODE generation.
- Package commands contains the various commands – stochastic simulation (single and replicated) and generation and solution of ODEs. These use classes from the package collection and are responsible for initializing the simulation/ODE solution and for graphical and file output of the resulting data.
- Package utils provides different utilities used by the commands, such as random variate samplers(exponential and discrete) and numerical ODE solver (fourth order Runge-Kutta).

See Figure 7.1 for an overview of the package structure.

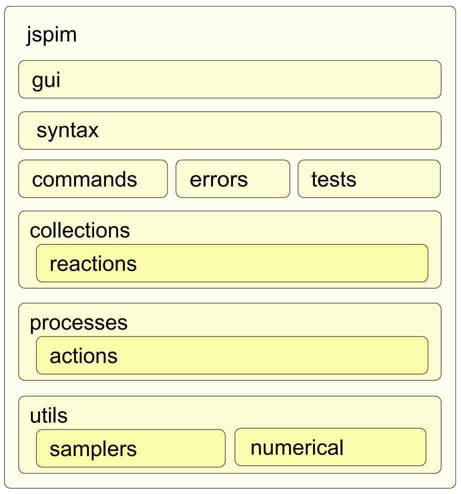


Figure 7.1: Overview of the main package structure of JSPiM. The packages are ordered by dependency – each package depends only on the packages bellow it.

7.1.1 Used libraries

JSPiM employs several external tools and libraries at different layers in the structure, including parsing, process representation and graphical output.

- *ANTLR*[29] is a language tool for building lexers, parsers and compilers. It is based on attribute grammars and automatically generates Java code. JSPiM uses the lexer and parser grammars to create $\mathcal{S}\pi$ abstract syntax tree (AST) and a list of commands from the textual representation and then a compiler to transform the AST to the below described representation. ANTLR is published under *BSD* license.
- *google-collections*[5] is a set of collection types, naturally extending the *Java Collections Framework*. JSPiM mainly uses the multiset and multimap collections (interfaces Multiset and Multimap and implementations HashMultiset and HashMultimap) to represent the structural congruence classes of processes and auxiliary data structures for simulation. *google-collections* is published under the Apache License 2.0 licence.
- *JFreeChart*[16] is a Java chart library. It is used to plot results of the different "executions" of $\mathcal{S}\pi$ processes (simulation, ODE solution, etc.). *JFreeChart* is published under the *LGPL* license.

7.2 Implementation details

We list the main important details of the implementation. We start at the bottom layer of parsing, then describe process representation, followed by the higher level collections and commands.

7.2.1 ANTLR grammars

Lexer
Parser

JSPiM uses three different ANTLR grammars. The Lexer defines individual tokens and eliminates white space and comments and Parser constructs an abstract syntax tree (internal ANTLR representation) corresponding to the syntactical structure of processes, environments and commands. It is defined with respect to the operator precedence to avoid unnecessary parentheses. See Appendix A.1 for the exact grammar used.

Compiler

The third grammar Compiler defines a tree-walker. This takes the abstract sytnax tree generated from the Parser and builds the JSPiM representation of processes, environment and commands. It also performs validity checks of the environment and provides mechanisms for error reporting. See Figure 7.2 for a diagram.

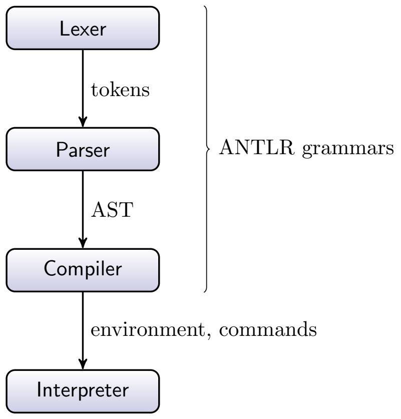


Figure 7.2: The role of ANTLR grammars in JSPiM. The Lexer splits the input character stream into tokens, the Parser constructs an abstract syntax tree and the Compiler builds the JSPiM representation of the processes, environments and commands. These are then passed to the interpreter.

7.2.2 Process representation

The used data structures representation closely follows the developed theory – we argue that this correspondence can almost directly serve as a proof of correctness. At the core is the representation of $\mathcal{S}\pi$ processes, naturally taken from the syntactical definition (and enhanced by some of the results concerning structural congruence) and adapted to the object oriented paradigm. The base class in the hierarchy is PiProcess, representing general processes from $\mathcal{P}$. It stores a link to the Environment object, representing the environments, as collections of ProcessDefinition objects. Representations of the functions ranging over processes and those inductively defined on their syntactical structure, such as $\mathrm{fn}(\cdot)$ and substitution $\cdot\{\widetilde{x} \mapsto \widetilde{\psi}\}$ are defined as abstract methods of the class PiProcess.

PiProcess

This base class is then extended by classes corresponding to the different syntactical constructs. See Figure 7.3 for an overview.

Zero

(i) The class Zero represents the zero process, trivially defining all the required functions.

Summation

(ii) The class Summation represents summations. It contains a collection of continuations prefixed with channel actions (ChannelContinuation) and with silent actions (DelayContinuation). Each continuation consists of the prefix action (class Action extended by ChannelAction and DelayAction) and the following process.

Parallel Composition

(iii) The class ParallelComposition represents trees of nested parallel compositions. The leaf processes are stored in a multiset (Multiset class from *google-collections*), as defined by the structural congruence.

ProcessID

(iv) The class ProcessID represents parametrized identifier instances. It stores the parameters in a list and has method getUnfolded() which returns the correct substitution instance from the right hand side of the corresponding defining equation in the environment.

Restriction

(v) The class Restriction represents chains of nested restrictions. It consists of a set of new channel definitions (class Channel, storing the name and the rate), as specified by the structural congruence, and of the restricted process.

Ideally, equality of the PiProcess objects would implement the structural congruence. Unfortunately, checking structural congruence for processes of π calculus (and so of $\mathcal{S}\pi$) is Graph Isomorphism complete, [23]. Therefore JSPiM implements only syntactical equality. This is reflected by the overriding of the equals() and hashCode() methods of the subclasses of PiProcess.

7.2.3 Higher level collections

A level higher from the processes are *collections*. These serve as bases for both the discrete and continuous semantics. See Figure 7.5 for an overview.

PiProcesses Collection

Reacting Collection

The ProcessesCollection is the main one from which the others derive. It consists of a set of new channel names and a multiset of processes and keeps auxiliary maps of possible channel communications. When a new process is added or an existing one removed, this map gets updated locally – only the affected channels change. The ReactingCollection derives from ProcessesCollection and supports methods for enumerating and sampling the reaction events. It keeps multimaps assigning events to each channel name and delay rate – channelEvents contains events for each channel, with the corresponding apparent rates. Similarly delayEvents keeps events for each silent action rate. It supports generation of the next reaction event (single iteration of the Algorithm 4) in the getNextEvent() method:

- First, a discrete distribution sampler chooses an event (the class utils.samplers.Discrete). The possible events are values of the two multimaps channelEvents and delayEvents.
- The chosen reaction is then applied by the applyReaction() method. This removes the reacting processes by calling removeProcess() from the ProcessesCollection and adds the continuations (two for a communication event and one for a silent one) by calling addProcess(), with possibly applying the substitution resulting from input action parameter binding.

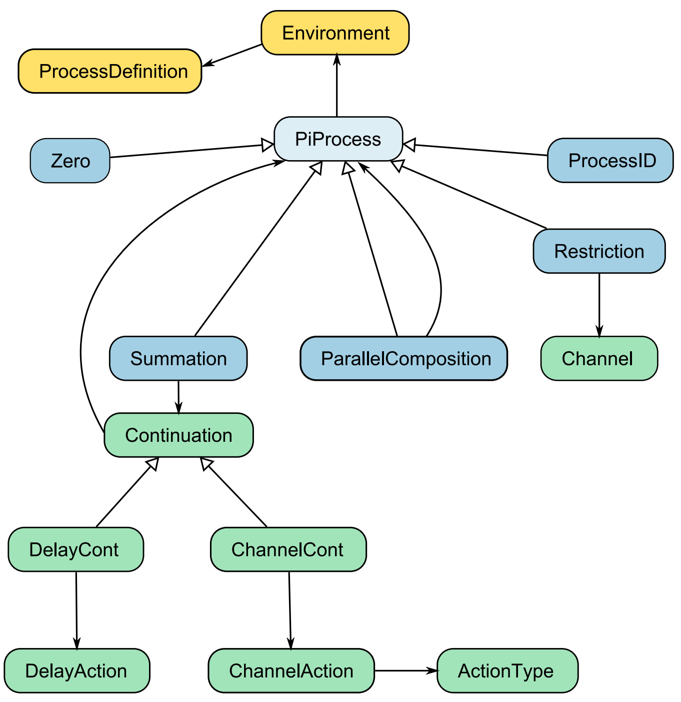


Figure 7.3: High level overview of $\mathcal{S}\pi$ process representation in the implementation. The base is the abstract class PiProcess, extended by classes representing the possible syntactical constructs, such as Zero, Summation, etc. Each process also keeps a reference to an Environment, containing a collection of defining equations (ProcessDefinition).

$P \in \mathcal{P}$	extends PiProcess
$\mathbf{0}$	Zero
$\alpha.P$	Continuation
$\sum_{i \in I} \alpha_i.P_i$	Summation
$P\|Q$	ParallelComposition
$A\langle\widetilde{\psi}\rangle$	ProcessID
$(\text{new}\, x@r)P$	Restriction

Figure 7.4: Representation of $\mathcal{S}\pi$processes in JSPiM.

- After this, both the reacting processes and their products are inspected to determine which channel names and delay rates have changed. For each changed channel name, the method collectChannelEvents() is called. In this method, the channel actions for the given name are retrieved and all possible combinations of input and output continuations are matched. For each pair, a new event is created with the probability calculated as the rate of the aggregated transition as given by the Theorem 3.3. Similarly, for each changed delay rate, the method collectDelayEvents() is called.
- Finally, a delay is generated from the exponential distribution (the utils.sampler.Exponential class) with parameter set to the sum of rates of all the events.

See Figure 7.6 for an overview of the two classes.

ODE Collection

The ODECollection handles the ODE generation for the continuous semantics of CGF. Given a system consisting of a top-level Process object and an Environment object, it generates the system of ODEs defining the continuous semantics in the following way

- First, prime processes are collected from the given process, using the algorithm $\widehat{\mathcal{P}}^*$ for CGF implemented in the method collectPrimeProcesses(), where the auxiliary function $\widehat{\mathcal{P}}_E$ corre-

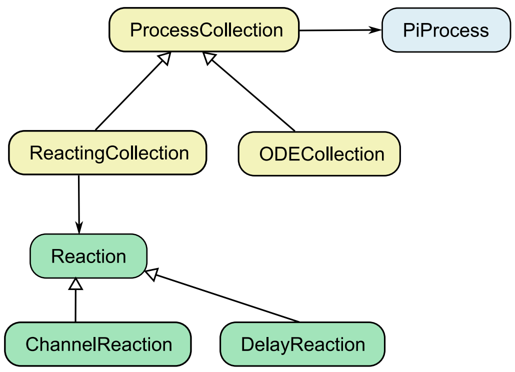


Figure 7.5: High level overview of the collections used for efficient simulation and ODE generation.

	ProcessesCollection
channels:	Set⟨Channel⟩
processes:	Multiset⟨PiProcess⟩
channelActions:	Map⟨PreAction, Multiset⟨ChannelContinuation⟩⟩
delayActions:	Map⟨Double, Multiset⟨DelayContinuation⟩⟩
■ addProcess()	
■ removeProcess()	
● getOccurences()	
	ReactingCollection
channelEvents:	Multimap⟨String, Event⟨Reaction⟩⟩
delayEvents:	Multimap⟨Double, Event⟨Reaction⟩⟩
■ applyReaction()	
● getNextEvent()	

Figure 7.6: Overview of the main fields and methods of the collections used for simulation in JSPiM.

sponds to the abstract method getPrimeProcesses() of the class PiProcess (defined in each of the subclasses). At this step the algorithm terminates if the given process or environment are not in CGF – an exception NotInCGFException is thrown (e.g. when trying to obtain prime processes from a Restriction).

- After this, the Enter and Exit sets are built in the method collectFormulas(), using the action maps from ProcessesCollection, and stored in the enterActions and exitActions respectively.
- In the method solveODEs(), a system of ODEs is defined in the object of class GeneratedODEs (extending the class SystemOfODEs used by the solver) using the enter and exit actions. The result from method rungeKutta() of the class utils.numerical.RungeKutta is returned.

See Figure 7.7 for an overview of the ODECollection class.

	ODECollection
enterActions:	List⟨List⟨Reaction⟩
exitActions:	List⟨List⟨Reaction⟩
■ collectPrimeProcesses()	
■ collectFormulas()	
● solveODEs()	

Figure 7.7: Overview of the main fields and methods of the collections used for ODE generation and solution in JSPiM.

$A(\widetilde{x}) \overset{\text{def}}{=} P$	ProcessDefinition
E	Environment
n_A	ProcessDef.getArity()
valid E	Environment.isValid()
$\text{fn}(P)$	PiProcess.getFreeNames()
$\text{bn}(P)$	PiProcess.getBoundNames()
$P\{\widetilde{x} \mapsto \widetilde{\psi}\}$	PiProcess.getSubstitution(Map⟨String,String⟩)
$P \equiv Q$	PiProcess.equals() (only some rules)
standard form	ProcessesCollection
$\uplus Q$	ProcessCollection.addProcess()
$\backslash Q$	ProcessCollection.removeProcess()
$P \xrightarrow{\alpha}$	PiProcess.getPossibleChannelActions()
$P \xrightarrow{\tau @ r}$	PiProcess.getPossibleDelayActions()
$P \notin \mathcal{P}_{\text{CGF}}$	NotInCGFException
$\widehat{\mathcal{P}}_E(P)$	PiProcess.getPrimeProcesses()
$P \# Q$	PiProcess.countOccurences(PiProcess Q)
$\text{Enter}_{-,S,E}$	ODECollection.enterActions
$\text{Exit}_{-,S,E}$	ODECollection.exitActions

Figure 7.8: Summary of representation of some of the $\mathcal{S}\pi$ definitions in JSPiM.

7.2.4 Commands

Command

Commands in JSPiM provide analysis of the discrete semantics of $\mathcal{S}\pi$ and the continuous semantics of CGF. All the commands extend the abstract class Command. The class Simulate responsible for stochastic simulation of $\mathcal{S}\pi$ systems implements the Algorithm 4, the class ODESolve implements ODE generation and solution for systems of CGF. The class MultipleSimulate performs replicated simulation of $\mathcal{S}\pi$ systems and calculates the transient mean and variance. See Figure 7.9 for an overview.

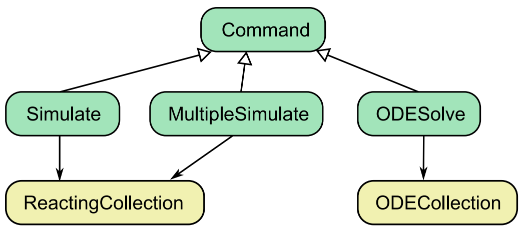


Figure 7.9: Overview of commands in JSPiM.

All the commands have some of their arguments in common. Each command analyses a system ($\mathcal{S}\pi$ or CGF). In JSPiM, we assume that every program defines a single environment, so the only explicit argument is the top-level process. Whether a simulation or ODE solution, the command requires the maximal time it should consider. To actually display any information, the commands require a list of *observed processes*. At each data point, populations of these are stored. Finally, in case of a file output, all three commands require a filename.

We briefly overview the three commands:

- The Simulate command implements the Algorithm 4. It uses ReactingCollection for each step of the iteration. It keeps executing the iteration step until the returned time from ReactingCollection.getNextEvent() reaches the given stopping time. As an argument it also takes an integer *data step*, specifying the number of steps between taking two consecutive data points, in order to provide some control over the size of the resulting data.

- The MultipleSimulate command repeats the simulation given number of times, using ReactingCollection in the same way as Simulate. It divides the time into intervals of given size (*time step*) and calculates mean and variance from data points in each of these intervals.

- The ODESolve command generates and solves the ODEs from the given CGF process, using ODECollection. It takes the time step as an argument, corresponding to the parameter h in Algorithm 3.

The arguments to the commands are summarized in the table below. See Figure 7.10 for a diagram overviewing the Simulate and ODESolve commands.

Simulate	process, time, data step, observed processes, filename
MultipleSimulate	process, time, replications, time step, data step, observed processes, filename
ODESolve	process, time, data step, inner step, observed processes, filename

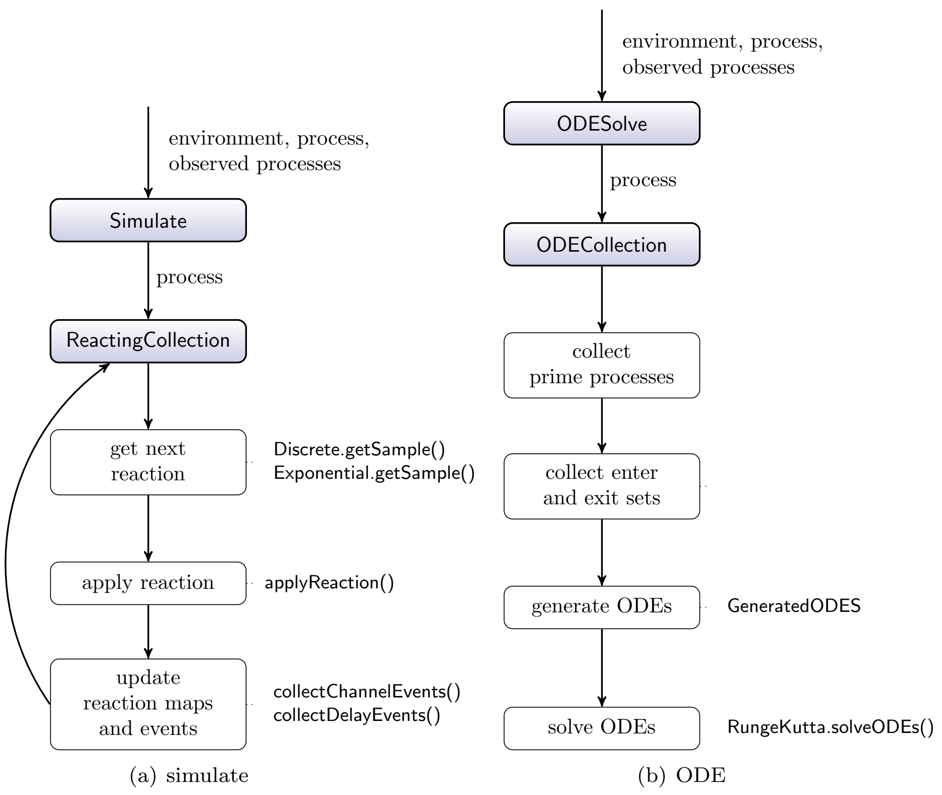


Figure 7.10: Summary of the simulation and ODE generation commands in JSPiM.

7.3 Spatial extension

The implementation of $\mathcal{L}\pi$ is analogous to how $\mathcal{L}\pi$ extends $\mathcal{S}\pi$. The main package of the extension is spatial, containing packages with the same name as those in the core. See Figre 7.11 for an overview.

- Package gui defines the new user interface components such as the simulation window with multiple locations.

- Package commands defines the new command for spatially simulating $\mathcal{L}\pi$ systems.

- Package collections defines the new collections used for spatial simulation of $\mathcal{L}\pi$ systems.
- Package syntax defines the extended lexer, parser and compiler to include the syntax of $\mathcal{L}\pi$.

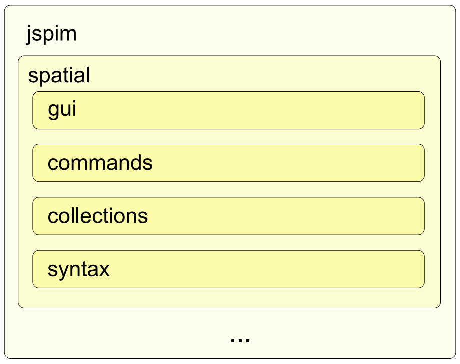


Figure 7.11: Overview of spatial packages in JSPiM.

The spatial extension defines location graphs in a class LocationGraph, which contains a map from location names to processes and the movement and volume functions. *LocationGraph*

Similar to the core implementation, the spatial extension defines higher level collections for efficient simulation and ODE generation. See Figure 7.12 for an overview.

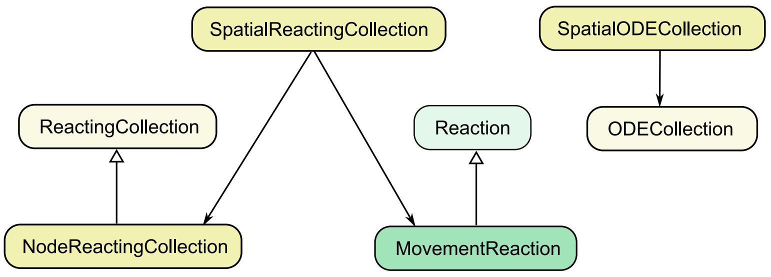


Figure 7.12: Spatial collections in JSPiM.

- The class SpatialReactingCollection implements an efficient single step of the Algorithm 5. It is using an extended ReactingCollection, the NodeReactingCollection for each location to obtain internal events and then adds movements, objects of class MovementReaction, to these for sampling. See Figure 7.13 and 7.14.
- The class SpatialODECollection implements ODE generation and solution. It uses ODECollection (and has the same method signature) for each location to obtain the enter and exit multisets for the inner transitions.

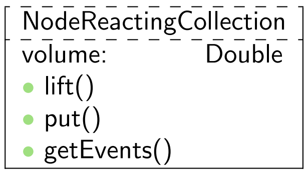


Figure 7.13: Overview of the main fields and methods of the NodeReactingCollection.

	SpatialReactingCollection
nodes:	Map⟨String, NodeReactingCollection⟩
m:	Map⟨Movement, Double⟩
🟥 getMovementEvents()	
● getNextEvent()	

Figure 7.14: Overview of the main fields and methods of the SpatialReactingCollection.

7.4 Testing

During the development, JSPiM has been tested in various ways. The package tests provides a way to test the consistency of the process representation. It contains test commands (subclasses of the Test class) which have the corresponding syntax defined in the Lexer and Parser. The class AreCongruent provides testing for the (limited) structural congruence implementation, allowing checks of the form $P \equiv Q$ and $P \not\equiv Q$, with syntax *Process* $\sim$ *Process* and *Process* $\sim$ *Process* respectively.

The class Reduces allows checks of the form $P \longrightarrow Q$ with syntax *Process*->*Process* and NotReduces allows to check whether $\mathit{Trans}(P) = \emptyset$ with syntax *Process*-|.

See Appendix C for a list of tests that have been used.

More complex testing has been done by comparing the simulation results with those of SPiM. We compared plots resulting from the models listed in Appendix B (and their scaled versions which lower the variance of the simulation traces). We also compared the ODE solutions with simulation averages.

7.5 Benchmarking

We ran several benchmarks based on the models in Appendix B and compared the simulation times (average of 5 runs) with those produced by SPiM. We can conclude that JSPiM gives comparable performance to SPiM. The tests were run on a system with Intel Core 2 Quad CPU Q8200 at 2.33GHz with 3GB RAM and MS Windows Vista operating system.

Model	JSPiM time	SPiM time
B.1	21 s	14 s
B.2	2 s	4 s
B.5	0 s	0 s
B.11	1 s	1 s
B.7.1	2 s	10 s

Figure 7.15: Comparison of running times of SPiM and JSPiM. The performance varies between models, due to the different implementation details of SPiM and JSPiM. Overall, the simulation times are in the same order and similarly depend on scaling.on scaling.

One thing to note is that JSPiM is not as memory efficient as SPiM. This is caused mainly by the overhead of the multisets and should be investigated before general release.

8 Evaluation and Future work

8.1 Multiset representation

In Chapter 3, we gave a standard definition of a variant of stochastic π calculus called $\mathcal{S}\pi$. We used the structural congruence to provide a multiset representation of processes. Although this is not novel in itself, we are not aware of any work where such representation was given explicitly for stochastic π calculus. This representation then proved to be useful for providing an efficient simulation algorithm and also for defining the continuous semantics. We add that it also gives slightly different view on stochastic π calculus – the processes do not correspond to individual molecules (as is often argued, e.g. in [12]) but instead they represent different species, with the multiset multiplicity corresponding to the population.

8.2 Continuous semantics

In Chapter 4, we defined and informally justified continuous semantics for $\mathcal{S}\pi$, by giving a *direct translation* from the calculus to a set of ordinary differential equations. We have shown that for a subset of $\mathcal{S}\pi$, the *Chemical Ground Form* (CGF), the resulting set of differential equations is finite. This differs from the approach taken in [7], where the author gives an indirect translation from CGF to ODEs via intermediate translation through chemical reaction equations. We have also shown efficient algorithms for obtaining this set of ODEs.

The continuous semantics certainly brings benefits of having multiple methods of analysis of $\mathcal{S}\pi$ models. We were able to apply it to various existing models in stochastic π calculus (some of them can be found in Appendix B).

8.3 Finiteness conditions

In Section 5.1, we gave two examples of $\mathcal{S}\pi$ systems with continuous semantics producing an infinite set of ODEs. These motivated formulation of two notions of finiteness of $\mathcal{S}\pi$ systems. One of the notions guarantees a finite set of ODEs of the continuous semantics of the system. The other, the *scalable finiteness*, guarantees that we can take any multiple of the system (i.e. arbitrarily increase the initial populations of processes) and get a constant number of ODEs.

We first gave a syntactical condition on $\mathcal{S}\pi$ that guarantees finiteness by considering a restriction that does not allow parallel composition in continuations. This condition didn't prove to be very useful, since the nature of models in Systems Biology usually requires creation of new processes; none of the examples in our collection of models satisfied this restriction. Using the two examples, we *informally* described how a system can be infinite and applied this reasoning to two models, Appendix B.1 and Appendix B.5, to reason that they are scalably finite, in order to carry out experiments in the following section.

This argument has severe limitations as it is not formal and only gives intuition about when $\mathcal{S}\pi$ systems are finite. To formalize it, more theory would have to be built to analyze the structure of $\mathcal{S}\pi$ environments – we believe that this lies outside of the scope of this project and leave the development for future work – our main aim was to translate $\mathcal{S}\pi$ models into CGF to provide the possibility of further analysis.

8.4 Relationship between the two semantics

In Section 5.2, we looked at comparing the continuous and discrete semantics. We are not aware of any investigation or results for stochastic π calculus in this area.

Inspired by work done for BioPEPA, we tried to confirm some convergence properties. Our hypothesis was that the continuous semantics of a system is a limit to its discrete semantics as it gets scaled. We experimentally confirmed this hypothesis on two examples. However, we found an example, a model of a gene repressilator, where this most likely doesn't hold, by showing that the simulation average (probably) converges to a different function than the continuous semantics. This argument can be supported by the fact that in [4], the author uses a model of a similar system to show limitations of the continuous semantics of *stochastic Concurrent Constraint Programming* and proposes the use of *hybrid systems*.

8.5 Spatial extension

In Chapter 6, we defined a spatial extension to $\mathcal{S}\pi$, named $\mathcal{L}\pi$, that allows to express models with static compartments of constant volume and processes moving between these. Both the syntax and semantics of $\mathcal{L}\pi$ are a direct extension of those for $\mathcal{S}\pi$. This allows convenient re-use of $\mathcal{S}\pi$ models and so supports the compositionality argument for our framework, as we illustrated on a simple SIR epidemic model from [7]. We gave an efficient algorithm as a basis for our implementation of $\mathcal{L}\pi$. We also presented an original example from plant biology demonstrating the flexibility of $\mathcal{L}\pi$.

We extended the continuous semantics to bring its benefits to $\mathcal{L}\pi$. We are not aware of any other spatial extension of stochastic π calculus that would provide continuous semantics. We have shown that the expressive power of $\mathcal{L}\pi$ is the same as that of $\mathcal{S}\pi$ but argued that it offers more succint modelling language.

8.6 Implementation

We developed a tool supporting $\mathcal{S}\pi$ and $\mathcal{L}\pi$. The tool, named *JSPiM*, is written in Java 1.5 programming language. It supports efficient simulation of $\mathcal{S}\pi$ systems. We used the multiset representation of $\mathcal{S}\pi$ processes to closely tie the implementation with the formally described algorithm, thus suggesting a proof of correctness. This is in contrast to the current state of the art stochastic π calculus implementation, the *Stochastic π machine* (SPiM) [32], where the authors prove correctness through an intermediate abstract machine [32]. JSPiM also implements the continuous semantics of systems in CGF, using representation close to the formalism and the efficient algorithms derived in Chapter 4. We are not aware of any implementation of the continuous semantics for CGF processes.

We also implemented the basic concepts of $\mathcal{L}\pi$ in JSPiM and demonstrated how the extension of $\mathcal{L}\pi$ from $\mathcal{S}\pi$ corresponds to the extension of the implementation.

Where applicable (i.e. in stochastic simulation of $\mathcal{S}\pi$ systems), we informally compared the performance of JSPiM with SPiM, giving comparable performance. This justifies future work for releasing JSPiM to the community, as it offers wider portability to SPiM and provides the continuous semantics of CGF as well as the spatial extension.

8.7 Collection of models

As a by product of our effort, we collected several example models in the Appendix B. These originate in the literature on stochastic π calculus applications in biology. We were able to enhance most of them (possibly after translation to CGF, using the ideas given in Chapter 5) with results from analysing the continuous semantics.

8.8 Future work

There are various different directions our work can be taken further. Firstly, the above mentioned outstanding challenges have to be tackled. These include

- *Formal justification of the continuous semantics.* The set of ODEs from continuous semantics of a $\mathcal{S}\pi$ system should be somehow formally related to the underlying CTMC, instead of the informal argument we presented here.

- *Formal finiteness results.* Our informal argument about the requirements for a system to be scalably finite should be formalized. Ideally, a necessary and sufficient condition should be found. Alternatively, it might be of interest to see whether the problem of determining if a given $\mathcal{S}\pi$ system is equivalent to a CGF system (one yielding the same CTMC) is decidable.

- *Relationship between the continuous and discrete semantics.* The investigation we carried out should be extended so a new hypothesis concerning the convergence of simulations to ODE solutions can be formulated and then proved. Ideally, similar argument to the one for BioPEPA in [14] would be formulated : a class of CTMCs could be found that has limiting behaviour solution to a set of ODEs and it could be proved that the CTMCs resulting from the discrete semantics belong to this class and the ODEs are precisely those generated by the continuous semantics. Most likely, such a class of CTMCs will not be found for the whole $\mathcal{S}\pi$, as we suggested by the repressilator example in Section 5.2. Therefore we could restrict our attention to some class of $\mathcal{S}\pi$ models where this could hold.

- *Models for $\mathcal{L}\pi$.* To fully justify the flexibility of $\mathcal{L}\pi$, real Systems Biology models should be formulated in $\mathcal{L}\pi$ and proven to bring insight into the biological system. This could involve extending the plant tissue example to make it more according to biological knowledge on the subject.

- *Extension of $\mathcal{L}\pi$.* We can look at whether there is a consistent way of extending $\mathcal{L}\pi$ to allow movement of complexes and evaluate the expressive power this adds to $\mathcal{L}\pi$ as compared to $\mathcal{S}\pi$. We can also restrict movement between locations and add explicit *movement actions*, which would have to be executed before a process moves.

- *Improved implementation.* Using the formal finiteness results, the JSPiM could implement the continuous semantics for the full $\mathcal{S}\pi$ and $\mathcal{L}\pi$. This would require finding an efficient algorithm for enumerating the prime processes of $\mathcal{S}\pi$ systems as well as the Enter and Exit multisets. This can be possibly achieved by considering some of the results about finiteness. Additionally, truncation of the set of prime processes can be defined, giving the possibility of continuous semantics even when the system is not finite.

  To improve the implementation of $\mathcal{L}\pi$, parametric definition of the location graphs and movement functions can be implemented for common families of graphs, such as grids, to avoid complicated descriptions of the models, such as the one for the plant tissue example where each member of the movement function has to be explicitly defined, see Appendix B.12.

  Also, JSPiM could be improved to implement some of the features of SPiM. For example, SPiM allows *typed* channels – that is specifying not only arity of the channel but also what type the arguments can be. We could include this in the implementation to allow better error reporting.

Last but not least, a release of JSPiM to the community would be valuable for obtaining feedback on the implementation as well as $\mathcal{L}\pi$ and the its possibilities for modelling.

Additional research can be carried out to use our developed formalisms. One obvious direction after we have defined the continuous semantics of $\mathcal{S}\pi$ would be to look at *hybrid semantics*, for example as has been done for the stochastic Concurrent Constraint Programming in [4]. This uses the formalism of hybrid automata, systems which combine continuous dynamics of with discrete stochastic control. We believe that the formalisms we presented in this project as well as the implementation can be valuable for research in this direction.

8.9 Conclusion

We provided a continuous and spatial extension of stochastic π calculus. We believe our contribution is valuable to the research community, by giving a new formalism, some insight into the relationship between continuous and discrete approaches to modelling and by providing a tool that can be used for analysing real examples from Systems Biology.

Bibliography

[1] R. Blossey, L. Cardelli, and A. Phillips. A compositional approach to the stochastic dynamics of gene networks. *Transactions on Computational Systems Biology*, IV(3939):99–122, 2006.

[2] L. Bortolussi. On the approximation of stochastic concurrent constraint programming by master equation. *Electron. Notes Theor. Comput. Sci.*, 220(3):163–180, 2008.

[3] L. Bortolussi and A. Policriti. Stochastic concurrent constraint programming and differential equations. *Electr. Notes Theor. Comput. Sci.*, 190(3):27–42, 2007.

[4] L. Bortolussi and A. Policriti. The importance of being (a little bit) discrete. *Electron. Notes Theor. Comput. Sci.*, 229(1):75–92, 2009.

[5] K. Bourrillion and J. Levy. google-collections. http://code.google.com/p/google-collections/.

[6] L. Cardelli. Abstract machines of systems biology. *Transactions on Computational Systems Biology*, III:145–168, 2005.

[7] L. Cardelli. From processes to odes by chemistry. In *IFIP TCS*, 2008.

[8] L. Cardelli, E. Caron, P. Gardner, O. Kahramanoğulları, and A. Phillips. A process model of actin polymerisation. In N. Cannata, E. Merelli, and I. Ulidowski, editors, *Proceedings of the Workshop "From Biology To Concurrency and back (FBTC 2008)", July 2008*, volume 229 of *Electronic Notes in Theoretical Computer Science*, pages 127–144, Reykjavik, Iceland, 2008. Elsevier.

[9] L. Cardelli, P. Gardner, and O. Kahramanoğulları. A process model of rho gtp-binding proteins in the context of phagocytosis. *Electron. Notes Theor. Comput. Sci.*, 194(3):87–102, 2008.

[10] L. Cardelli and G. Zavattaro. On the computational power of biochemistry. In *AB '08: Proceedings of the 3rd international conference on Algebraic Biology*, pages 65–80, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] F. Ciocchetta and M. L. Guerriero. Modelling biological compartments in bio-pepa. *Electron. Notes Theor. Comput. Sci.*, 227:77–95, 2009.

[12] F. Ciocchetta and J. Hillston. Bio-pepa: a framework for the modelling and analysis of biological systems, 2008. Theoretical Computer Science.

[13] W. Cohen. *A Computer Scientist's Guide to Cell Biology.* Springer, 2007.

[14] N. Geisweiller, J. Hillston, and M. Stenico. Relating continuous and discrete PEPA models of signalling pathways. *Theor. Comput. Sci.*, 404(1-2):97–111, 2008.

[15] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104(9):1876–1889, March 2000.

[16] D. Gilbert. Jfreechart. http://www.jfree.org/jfreechart/.

[17] D. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[18] G. Grimmett and D. Stirzaker. *Probability and Random Processes.* Oxford University Press, 2001.

[19] A. Gecioueur. Modelling aba signal transduction in plants using a stochastic process calculus, 2008.

[20] J. Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, Sept. 2005. IEEE Computer Society Press.

[21] C. Y. Huang and J. E. Ferrell. Ultrasensitivity in the mitogen-activated protein kinase cascade. *Proc Natl Acad Sci U S A*, 93(19):10078–10083, September 1996.

[22] M. John, R. Ewaki, and A. Uhrmacher. A spatial extension to the π calculus. In *Proceedings on the First Workshop "From Biology to concurrency and back"*, volume 194, 2007.

[23] V. Khomenko and R. Meyer. Checking π-calculus structural congruence is graph isomorphism complete. Technical Report CS-TR: 1100, School of Computing Science, Newcastle University, 2008. 20 pages.

[24] C. Kuttler and J. Niehren. Gene regulation in the pi calculus: simulating cooperativity at the lambda switch. *Transactions on Computational Systems Biology VII*, 4230:24–55, 2006.

[25] M. Kwiatkowski and I. Stark. The continuous π-calculus: A process algebra for biochemical modelling. In *Computational Methods in Systems Biology: Process of the Sixth International Conference CMSB 2008*, number 5307 in Lecture Notes in Computer Science, pages 103–122. Springer-Verlag, 2008.

[26] T. Lu, D. Volfson, L. Tsimring, and J. Hasty. Cellular growth and division in the gillespie algorithm. *Systems Biology*, 1(1):121–128, 2004.

[27] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[28] V. Muganathan, A. Phillips, and Maria G. Vigliotti. Bam: Bioambient machine. In *ACSD 08*, To appear.

[29] T. Parr. Antlr. `http://www.antlr.org/`.

[30] A. Phillips. Examples in spim. `http://research.microsoft.com/en-us/projects/spim/examples.pdf`.

[31] A. Phillips and L. Cardelli. A graphical representation for the stochastic pi-calculus. In *Proceedings BioConcur 2005*, 2005.

[32] A. Phillips and L. Cardelli. Efficient, correct simulation of biological processes in stochastic π-calculus. *Proceedings of Computational Methods in Systems Biology*, pages 184–199, 2007.

[33] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(7), 1995.

[34] C. Priami and P. Quaglia. Beta binders for biological interactions. In *Computational Methods in Systems Biology*, volume 3082/2005, pages 20–33. Springer, 2005.

[35] A. Regev, E. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science, Special Issue on Computational Methods in Systems Biology*, 325(1):141–167, 2004.

[36] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. *Pac Symp Biocomput*, pages 459–470, 2001.

[37] S. Ross. *Introduction to Probability Models.* Academic Press, 2007.

[38] N. Segata and E. Blanzieri. Stochastic pi-calculus modelling of multisite phosphorylation based signalling: The pho pathway in sccharomyces cerevisiae. *Lecture Notes in Computer Science*, 2008.

[39] C. Versari and N. Busi. Stochastic simulation of biological systems with dynamical compartment structure. *CMSB*, pages 80–95, 2007.

[40] D. Wilkinson. *Stochastic Modelling for Systems Biology.* Chapman & Hall/CRC, 2006.

[41] O. Wolkenhauer, M. Ullah, W. Kolch, and K.-H. Cho. Modelling and simulation of intracellular dynamics: Choosing an appropriate framework. *IEEE Transactions on NanoBioscience*, 3:200–207, 2004.

A JSPiM

A.1 Language definition

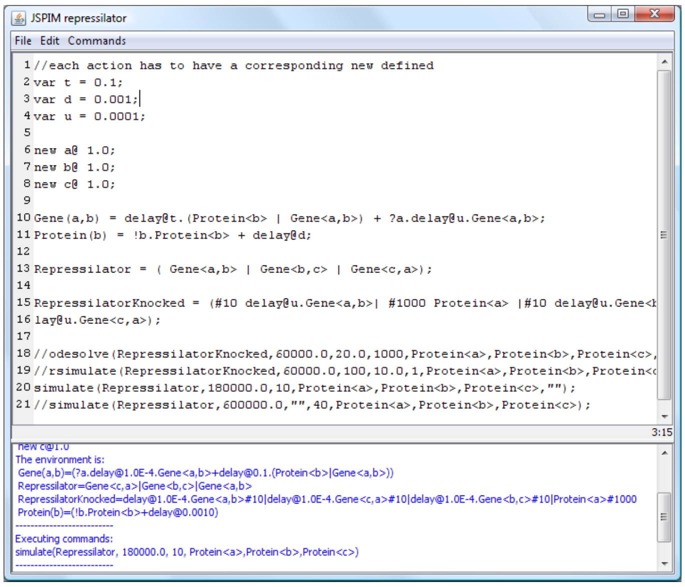
A.1.1 Core

$$\begin{aligned}
\mathit{Program} &:= \mathit{RateDefinition}^{*}\ \mathit{ChannelDefinition}^{*}\ \mathit{ProcessDefinition}^{*}\ \mathit{Command}^{*}\\
\mathit{RateDefinition} &:= \texttt{var}\ \mathit{rateID}\ =\ \mathit{value}\ ;\\
\mathit{ChannelDefinition} &:= \texttt{new}\ \mathit{channelID}\ @\ (\mathit{value} \mid \mathit{rateID})\ ;\\
\\
\mathit{ProcessDefinition} &:= \mathit{processID}\ \{(\ \mathit{NameList}\ )\}{=}\mathit{Process}\ ;\\
\mathit{Process} &:= 0\\
&\quad |\mathit{Action}.\mathit{Process}+\ \cdots\ +\mathit{Action}.\mathit{Process}\\
&\quad |\mathit{Process}\mid\mathit{Process}\\
&\quad |\#\mathit{int}\ \mathit{Process}\\
&\quad |(\texttt{new}\ \mathit{Channel}\ (,\ \mathit{Channel})^{*}\ )(\ \mathit{Process}\ )\\
&\quad |\mathit{ProcessID}\ \{<\ \mathit{NameList}\ >\}\\
\mathit{Channel} &:= \mathit{channelID}\ @\ (\mathit{value} \mid \mathit{rateID})\\
\mathit{Action} &:= !(\mathit{channelID} \mid \mathit{variableID})\{<\mathit{NameList}>\}\\
&\quad \mid\ ?(\mathit{channelID} \mid \mathit{variableID})\{(\mathit{NameList})\}\\
\mathit{NameList} &:= (\mathit{variableID} \mid \mathit{channelID})\ (,\ (\mathit{variableID} \mid \mathit{channelID}))^{*}\\
\\
\mathit{Command} &:= \texttt{simulate(}\ \mathit{Process}\ ,\ \mathit{value}\ ,\ \mathit{int}\ ,\ \mathit{ProcessList}\ ,\ \mathit{filename}\ );\\
&\quad \mid \texttt{rsimulate(}\ \mathit{Process}\ ,\ \mathit{value}\ ,\ \mathit{int}\ ,\ \mathit{value}\ ,\ \mathit{int}\ ,\ \mathit{ProcessList}\ ,\ \mathit{filename}\ );\\
&\quad \mid \texttt{odesolve(}\ \mathit{Process}\ ,\ \mathit{value}\ ,\ \mathit{value}\ ,\ \mathit{int}\ ,\ \mathit{ProcessList}\ ,\ \mathit{filename}\ );\\
\mathit{ProcessList} &:= \mathit{Process}\ (,\mathit{Process})^{*}
\end{aligned}$$

A.1.2 Spatial extension

$$\begin{aligned}
\mathit{Program} &:= \mathit{RateDefinition}^{*}\ \mathit{ChannelDefinition}^{*}\ \mathit{ProcessDefinition}^{*}\ \mathit{GraphDefinition}^{*}\\
&\qquad \mathit{Command}'^{*}\\
\mathit{GraphDefinition} &:= \texttt{spatial}\ \mathit{GraphID}\ = \{\mathit{NodesList}\ \mathit{MovementDefinition}\ \}\\
\mathit{NodesList} &:= [\mathit{nodeID}{:}\mathit{Process}\ (,\ \mathit{nodeID}{:}\mathit{Process})^{*}];\\
\mathit{MovementDefinition} &:= (\texttt{m}(\mathit{nodeID},\mathit{nodeID},\mathit{Process}) = \mathit{value};)^{*}\\
\mathit{Command}' &:= \mathit{Command}|\\
&\qquad \texttt{spatialSimulate}(\ \mathit{graphID}\ ,\ \mathit{value}\ ,\ \mathit{int}\ ,\ \mathit{ProcessList}\ ,\ \mathit{filename}\ );
\end{aligned}$$

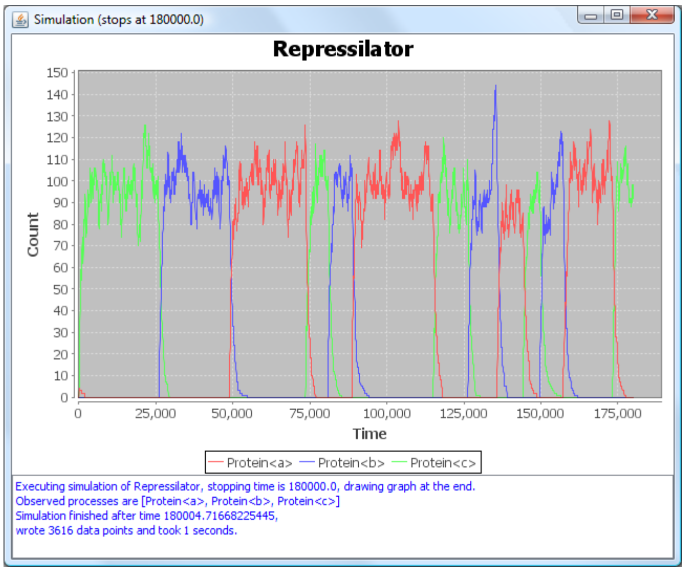
A.2 Screenshots



```
//each action has to have a corresponding new defined
var t = 0.1;
var d = 0.001;
var u = 0.0001;

new a@ 1.0;
new b@ 1.0;
new c@ 1.0;

Gene(a,b) = delay@t.(Protein<b> | Gene<a,b>) + ?a.delay@u.Gene<a,b>;
Protein(b) = !b.Protein<b> + delay@d;

Repressilator = ( Gene<a,b> | Gene<b,c> | Gene<c,a>);

RepressilatorKnocked = (#10 delay@u.Gene<a,b>| #1000 Protein<a> |#10 delay@u.Gene<
lay@u.Gene<c,a>);

//odesolve(RepressilatorKnocked,60000.0,20.0,1000,Protein<a>,Protein<b>,Protein<c>,
//rsimulate(RepressilatorKnocked,60000.0,100,10.0,1,Protein<a>,Protein<b>,Protein<
simulate(Repressilator,180000.0,10,Protein<a>,Protein<b>,Protein<c>,"");
//simulate(Repressilator,600000.0,"",40,Protein<a>,Protein<b>,Protein<c>);
```

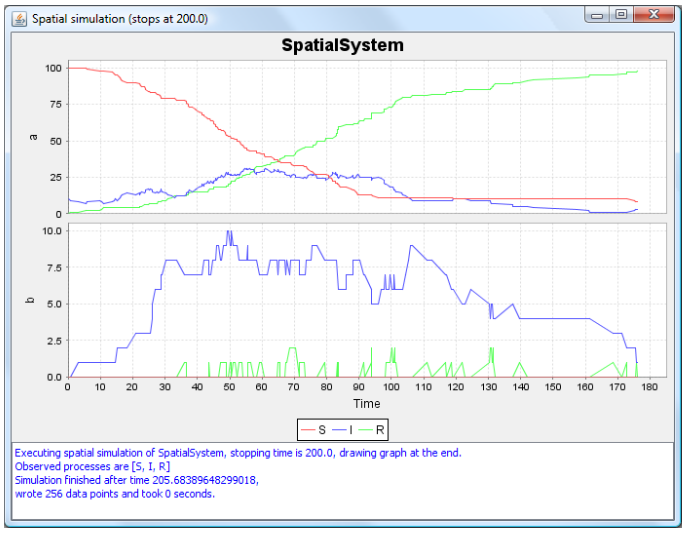
```
 new c@1.0
The environment is:
 Gene(a,b)=(?a.delay@1.0E-4.Gene<a,b>+delay@0.1.(Protein<b>|Gene<a,b>))
 Repressilator=Gene<c,a>|Gene<b,c>|Gene<a,b>
 RepressilatorKnocked=delay@1.0E-4.Gene<a,b>#10|delay@1.0E-4.Gene<c,a>#10|delay@1.0E-4.Gene<b,c>#10|Protein<a>#1000
 Protein(b)=(!b.Protein<b>+delay@0.0010)
-------------------------
Executing commands:
simulate(Repressilator, 180000.0, 10, Protein<a>,Protein<b>,Protein<c>)
-------------------------
```

Figure A.1: Screenshot from JSPiM.



Executing simulation of Repressilator, stopping time is 180000.0, drawing graph at the end.
Observed processes are [Protein<a>, Protein<b>, Protein<c>]
Simulation finished after time 180004.71668225445,
wrote 3616 data points and took 1 seconds.

Figure A.2: Screenshot from the simulation window.



Executing spatial simulation of SpatialSystem, stopping time is 200.0, drawing graph at the end.
Observed processes are [S, I, R]
Simulation finished after time 205.68389648299018,
wrote 256 data points and took 0 seconds.

Figure A.3: Screenshot from the spatial simulation window.

B Collection of basic examples

This is a collection of examples of models on which the provided tool was tested. Most of these have origins in the SPiM example library [30].

We present source code of JSPiM and results obtained from running the given commands, with the graphs taken from the GNUPlot output of JSPiM.

Because the syntax of SPiM and JSPiM is slightly different, we translated the models manually. This resulted most of the time to only string replacement.

B.1 Circadian clock

Translated from a model in the SPiM library[30].

B.1.1 Model

```
var drA = 1.0s;
var drR = 0.02;
var dA = 0.1;
var dR = 0.01;
var tA = 4.0;
var tR = 0.001;
var tA' =40.0;
var tR' =2.0;
var trA = 1.0;
var trR = 0.1;

new bind@100.0;
new pA@10.0;
new pR@10.0;

DNA_A = delay@tA.(RNA_A| DNA_A) + ?pA(u).DNA_A'<u>;
DNA_A'(u) = delay@tA'.(RNA_A | DNA_A'<u>) + !u.DNA_A;
RNA_A  = delay@trA.(A|RNA_A) + delay@drA;
A = (new u@1.0 uA@10.0 uR@100.0)
    (!pA<uA>.?uA.A + !pR<uR>.?uR.A
      + delay@dA + !bind<u>.A_Bound<u>);
A_Bound(u) = delay@dA.!u + ?u.A;
DNA_R = delay@tR.(RNA_R| DNA_R) + ?pR(u).DNA_R'<u>;
DNA_R'(u) = delay@tR'.(RNA_R | DNA_R'<u>) + !u.DNA_R;
RNA_R  = delay@trR.(R|RNA_R) + delay@drR;
R = delay@dR + ?bind(u).R_Bound<u>;
R_Bound(u) = ?u.R + delay@dR.!u;
Clock = (DNA_A|DNA_R);

simulate(Clock,800.0,60,RNA_A,RNA_R,A,R,A_Bound,"tmp/clock.csv");
```

B.1.2 Results

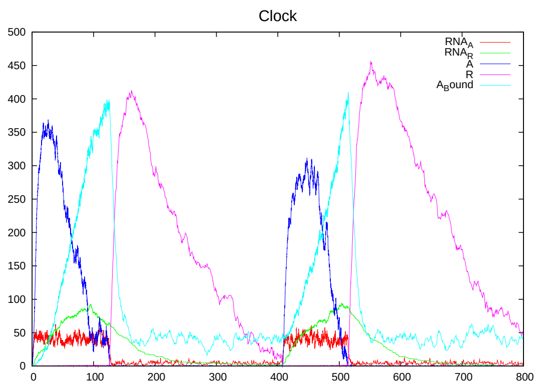


Figure B.1: Sample simulation of the Circadian clock model.

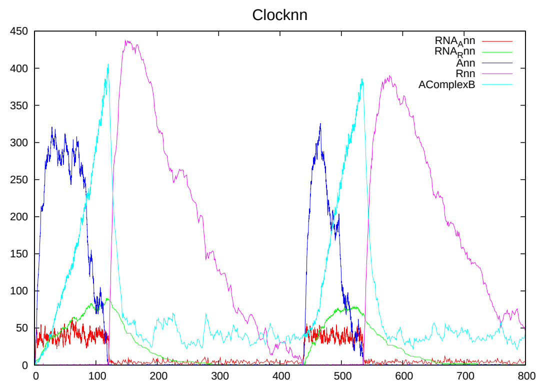
B.2 Circadian clock in CGF

Translation of the above model to CGF (using ideas described in the Chapter 5).

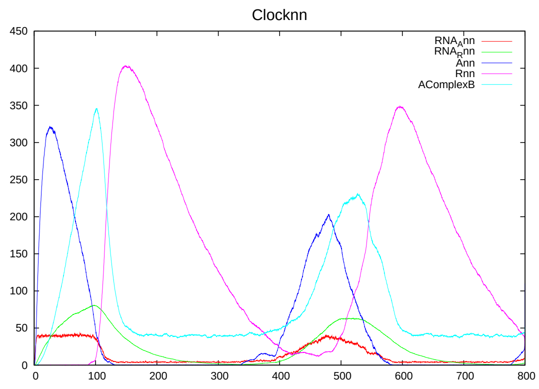
B.2.1 Model

```
var drA = 1.0;
var drR = 0.02;
var dA = 0.1;
var dR = 0.01;
var tA = 4.0;
var tR = 0.001;
var tA' =40.0;
var tR' =2.0;
var trA = 1.0;
var trR = 0.1;

new bind@100.0;
new pA@10.0;
new pR@10.0;

DNA_Ann = delay@tA.(RNA_Ann|DNA_Ann) + ?pA;
DNA_Rnn = delay@tR.(RNA_Rnn|DNA_Rnn) + ?pR;

RNA_Ann = delay@trA.(Ann|RNA_Ann) + delay@drA;
RNA_Rnn = delay@trR.(Rnn|RNA_Rnn) + delay@drR;

Ann = !pA.AComplexA + !pR.AComplexR + !bind.AComplexB + delay@dA;
Rnn = delay@dR + ?bind;

AComplexA = (delay@tA'.(RNA_Ann|AComplexA))+delay@10.0.(DNA_Ann|Ann);
AComplexR = (delay@tR'.(RNA_Rnn|AComplexR))+delay@100.0.(DNA_Rnn|Ann);
AComplexB = delay@dA.delay@1.0.Rnn + delay@dR.delay@1.0.Ann;

Clocknn = (DNA_Ann|DNA_Rnn);

odesolve(Clocknn,800.0,0.1,2000,RNA_Ann,RNA_Rnn,Ann,Rnn,AComplexB,"");
rsimulate(Clocknn,500.0,40,0.1,20,RNA_Ann,RNA_Rnn,Ann,Rnn,AComplexB,"");
simulate(Clocknn,800.0,40,RNA_Ann,RNA_Rnn,Ann,Rnn,AComplexB,"");
```

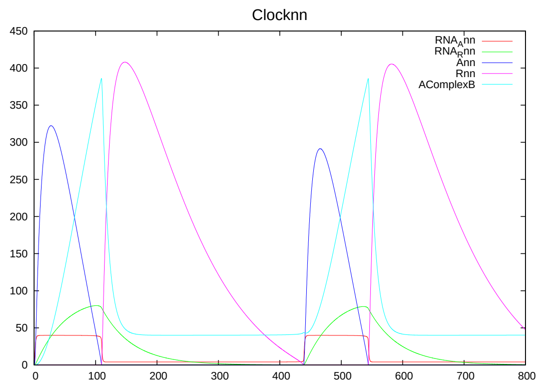
B.2.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.2: Sample simulation of the CGF translation of the circadian clock model.

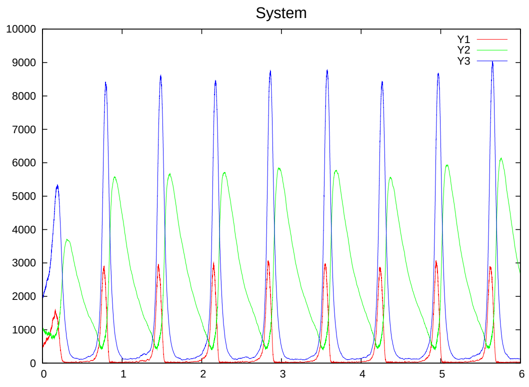
B.3 Oregonator 1

Translated from a model in the SPiM library[30], also present in [17].

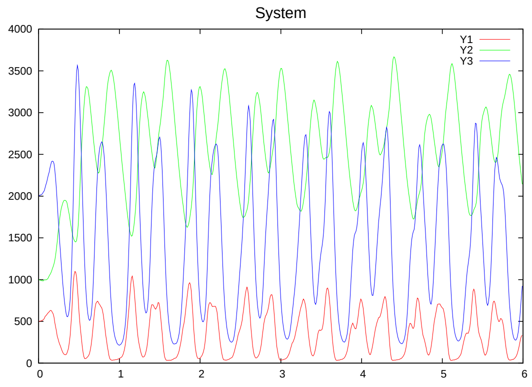
B.3.1 Model

```
new c1@2.0;
new c2@0.1;
new c3@104.0;
new c4@0.008;
new c5@26.0;

X1 = ?c1.X1;
X2 = ?c3.X2;
X3 = ?c5.X3;
Y1 = !c2 + !c3.(Y3|Y1|Y1) + !c4 + ?c4;
Y2 = !c1.Y1 + ?c2;
Y3 = !c5.Y2;

System = (X1|X2|X3|#500 Y1|#1000 Y2|#2000 Y3);


odesolve(System,6.0,0.005,10,Y1,Y2,Y3,"");

simulate(System,6.0,60,Y1,Y2,Y3,"");
rsimulate(System,6.0,20,0.01,1,Y1,Y2,Y3,"");
```

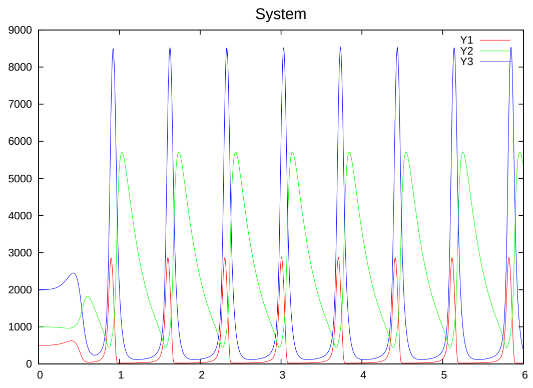
B.3.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.3: Results from the Oregonator 1 model.

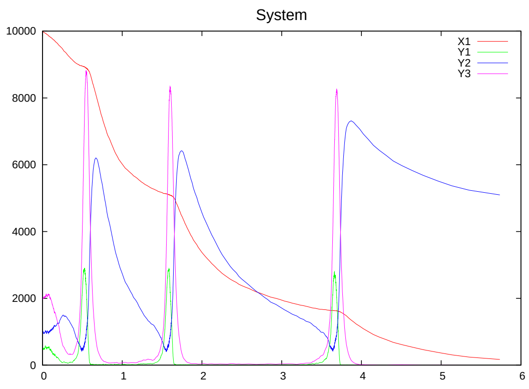
B.4 Oregonator 2

Translated from a model in the SPiM library[30].

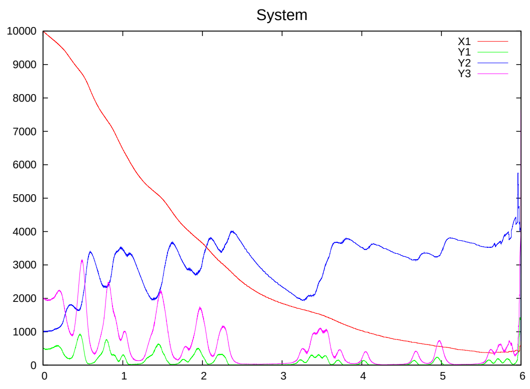
B.4.1 Model

```
new c1@0.0002;
new c2@0.1;
new c3@104.0;
new c4@0.008; //(* 0.016 / 2 *)
new c5@26.0;

X1 = ?c1;
X2 = ?c3. X2;
X3 = ?c5. X3;
Y1 =
   !c2
  + !c3. (Y3 | Y1 | Y1)
  + !c4
  + ?c4;
Y2 =
   !c1. Y1
  + ?c2;
Y3 = !c5. Y2;

System = (#10000 X1 | X2 | X3 |
          #500   Y1 | #1000 Y2 | #2000 Y3);
odesolve(System,6.0,0.0004,10,X1,Y1,Y2,Y3);
rsimulate(System,6.0,"",100,0.01,200,X1,Y1,Y2,Y3);
simulate(System,6.0,"",200,X1,Y1,Y2,Y3);
```

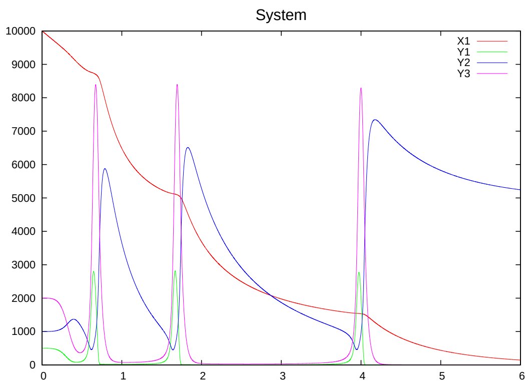
B.4.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.4: Results from the Oregonator 2 model.

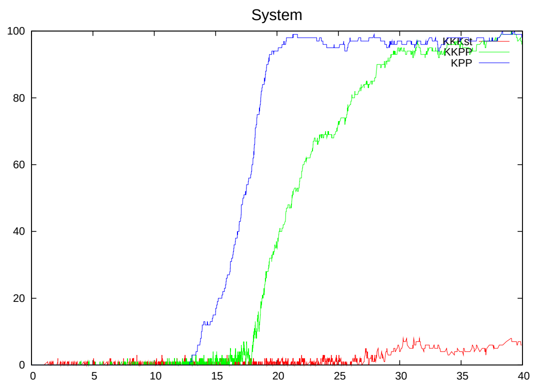
B.5 MAPK 1

Translated from a model in the SPiM library[30].

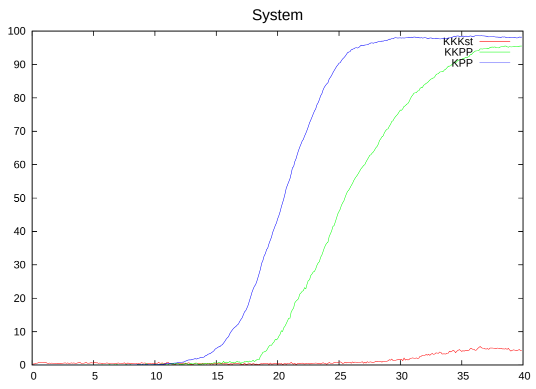
B.5.1 Model

```
var ra = 1.0;
var rd = 1.0;
var rk = 1.0;

new a1@ra;
new a2@ra;
new a3@ra;
new a4@ra;
new a5@ra;
new a6@ra;
new a7@ra;
new a8@ra;
new a9@ra;
new a10@ra;

 E1 = (new k1@rk d1@rd)(!a1<d1,k1>.(?d1.E1 + ?k1.E1));
 E2 = (new k2@rk d2@rd)(!a2<d2,k2>.(?d2.E2 + ?k2.E2));
 KKK = ?a1(d,k).(!d.KKK + !k.KKKst);
 KKKst = (new d3@rd k3@rk d5@rd k5@rk)(
    ?a2(d,k).(!d.KKKst + !k.KKK) + !a3<d3,k3>.(?d3.KKKst + ?k3.KKKst)
      + !a5<d5,k5>.(?d5.KKKst + ?k5.KKKst));
 KK = ?a3(d,k).( !d.KK + !k.KKP);
 KKP = ?a4(d,k).(!d.KKP + !k.KK)  + ?a5(d,k).(!d.KKP + !k. KKPP);
 KKPP = (new d7@rd k7@rk d9@rd k9@rk)(
    ?a6(d,k). ( !d. KKPP + !k. KKP)
  + !a7<d7,k7>. ( ?d7. KKPP + ?k7. KKPP)
  + !a9<d9,k9>. ( ?d9. KKPP + ?k9. KKPP));
 K = ?a7(d,k).(!d.K + !k.KP);
 KP = ?a8(d,k).( !d.KP + !k.K) + ?a9(d,k).( !d.KP + !k.KPP);
 KPP = ?a10(d,k).(!d.KPP + !k.KP);
 KKPase = (new d4@rd k4@rk d6@rd k6@rk)(
    !a4<d4,k4>.(?d4.KKPase + ?k4.KKPase) + !a6<d6,k6>. ( ?d6.KKPase + ?k6.KKPase));
 KPase = (new d8@rd k8@rk d10@rd k10@rk)
 (!a8<d8,k8>.( ?d8.KPase + ?k8.KPase) + !a10<d10,k10>.( ?d10.KPase + ?k10.KPase));

System =  ( E1 | #10 KKK | #100 KK | #100 K | E2 | KKPase | KPase );

simulate(System,40.0,1,KKKst,KKPP,KPP,"");
rsimulate(System,40.0,20,0.1,1,KKKst,KKPP,KPP,"");
```

B.5.2 Results



(a) simulation



(b) replicated simulation

Figure B.5: Simulation of the MAPK model.

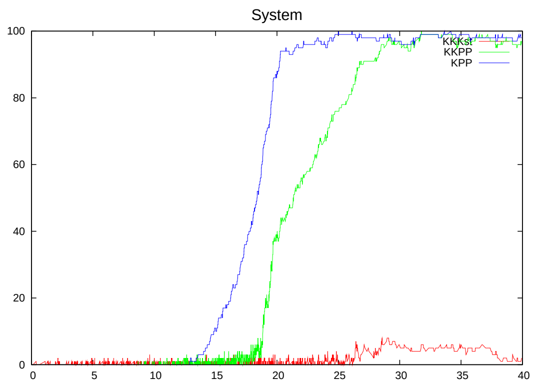
B.6 MAPK 1 in CGF

Translation of the above model to CGF.

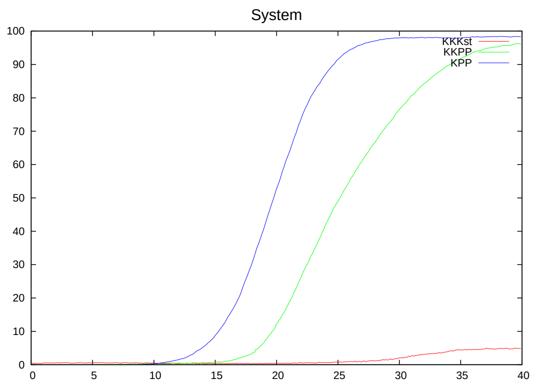
B.6.1 Model

```
var ra = 1.0;
var rd = 1.0;
var rk = 1.0;

new a1@ra;
new a2@ra;
new a3@ra;
new a4@ra;
new a5@ra;
new a6@ra;
new a7@ra;
new a8@ra;
new a9@ra;
new a10@ra;

 E1 = !a1.E1Complex;
 KKK = ?a1;
 E1Complex = delay@rd.(E1|KKK) + delay@rk.(E1|KKKst);

 E2 = !a2.E2Complex;
 E2Complex = delay@rk.(E2|KKK) + delay@rd.(E2|KKKst);
 KKKst = ?a2 + !a3.KKKstComplex1 + !a5.KKKstComplex2;

 KKKstComplex1 = delay@rd.(KKKst|KK) + delay@rk.(KKKst|KKP);
 KKKstComplex2 = delay@rd.(KKKst|KKP) + delay@rk.(KKKst|KKPP);
 KK = ?a3;
 KKP = ?a4 + ?a5;
 KKPP = ?a6 + !a7.KKPPComplex1 + !a9.KKPPComplex2;
 KKPPComplex1 = delay@rd.(KKPP|K) + delay@rk.(KKPP|KP);
 KKPPComplex2 = delay@rd.(KKPP|KP) + delay@rk.(KKPP|KPP);
 K = ?a7;
 KP = ?a8 + ?a9;
 KPP = ?a10;
 KKPase = !a4.KKPaseComplex1 + !a6.KKPaseComplex2;
 KKPaseComplex1 = delay@rd.(KKPase|KKP) + delay@rk.(KKPase|KK);
 KKPaseComplex2 = delay@rd.(KKPase|KKPP) + delay@rk.(KKPase|KKP);
 KPase = !a8.KPaseComplex1 + !a10.KPaseComplex2;
 KPaseComplex1 = delay@rd.(KPase|KP) + delay@rk.(KPase|K);
 KPaseComplex2 = delay@rd.(KPase|KPP) + delay@rk.(KPase|KP);

System =  ( E1 | #10 KKK | #100 KK | #100 K | E2 | KKPase | KPase );

simulate(System,40.0,1,KKKst,KKPP,KPP,"");
odesolve(System,40.0,0.1,10,KKKst,KKPP,KPP,"");
rsimulate(System,40.0,100,0.1,1,KKKst,KKPP,KPP,"");
```

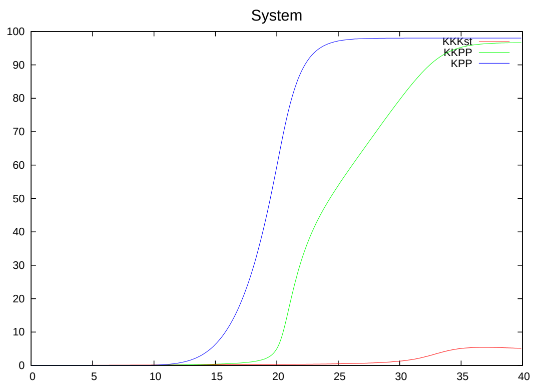
B.6.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.6: Results from the MAPK1 (in CGF) model.

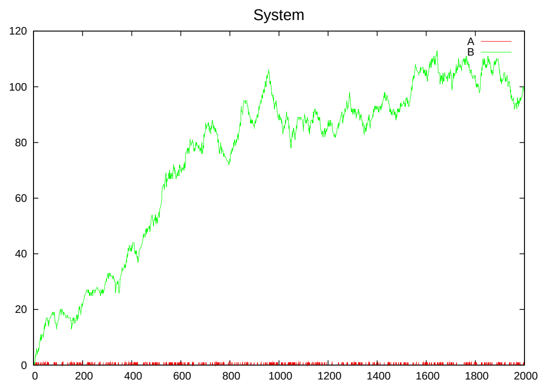
B.7 Bistable

Translated from a model in the SPiM library[30].

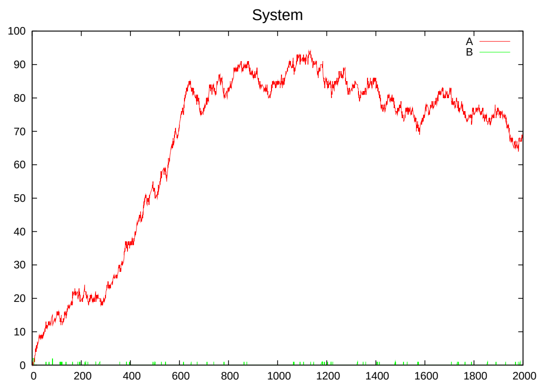
B.7.1 Model

```
var tA = 0.20;
var dA = 0.002;
var tB = 0.37;
var dB = 0.002;
var dAB = 0.53;
var unbind = 0.42;
var tB' = 0.027;

new bind@0.72;
new inhibit@0.19;

 Aa = delay@tA.( A | Aa );
 A = (new u@unbind)(
     delay@dA
  + !bind<u>.A_B<u>
  + !inhibit<u>.A_b<u>);
 A_b(u) = ?u.A;
 A_B(u) = delay@dAB;
 Bb =
   delay@tB.( B | Bb )
  + ?inhibit(u).Bb_A<u>;
 Bb_A(u) =!u.Bb + delay@tB'.( B | Bb_A<u> );
 B = ?bind(u).B_A<u> + delay@dB;
 B_A(u) = 0;
System =  (Aa | Bb);

simulate(System,10000.0,1,A,B,"");
rsimulate(System,2000.0,100,1.0,1,A,B,"");
```

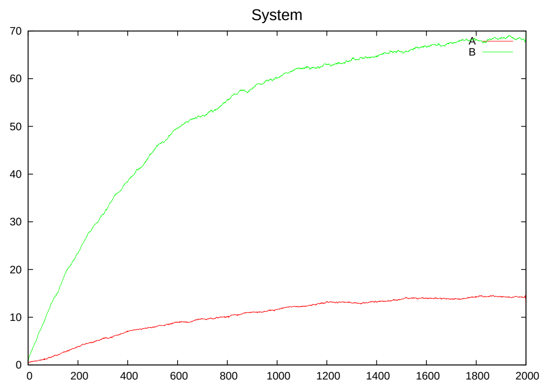
B.7.2 Results



(a) simulation



(b) simulation alternative



(c) replicated simulation

Figure B.7: Results from the Bistable model.

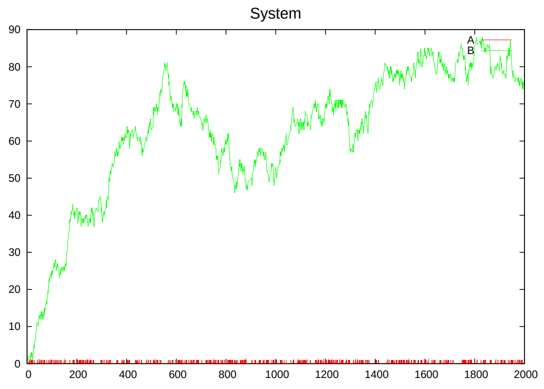
B.8 Bistable in CGF

Translation of the above model to CGF.

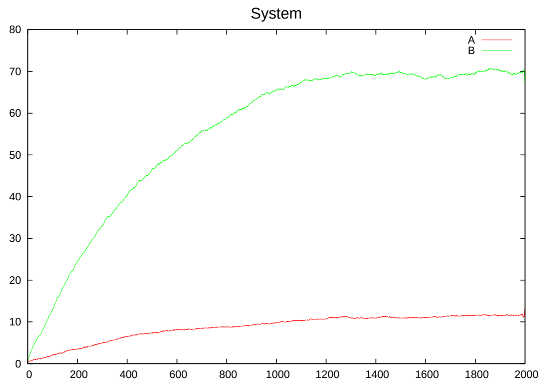
B.8.1 Model

```

var tA = 0.20;
var dA = 0.002;
var tB = 0.37;
var dB = 0.002;
var dAB = 0.53;
var unbind = 0.42;
var tB' = 0.027;


new bind@0.72;
new inhibit@0.19;


 Aa = delay@tA.( A | Aa );

 A =
     delay@dA
  + !bind.AComplexBind
  + !inhibit.AComplexInhibit;
 AComplexBind = (delay@dAB);
 AComplexInhibit = delay@unbind.(A|Bb) + delay@tB'.(B|AComplexInhibit);


 B = ?bind + delay@dB;

 Bb =
   delay@tB.( B | Bb )
  + ?inhibit;





System =  (Aa | Bb);

odesolve(System,2000.0,1.0,1000,A,B,"");
simulate(System,2000.0,1,A,B,"");
rsimulate(System,2000.0,100,1.0,1,A,B,"");

```

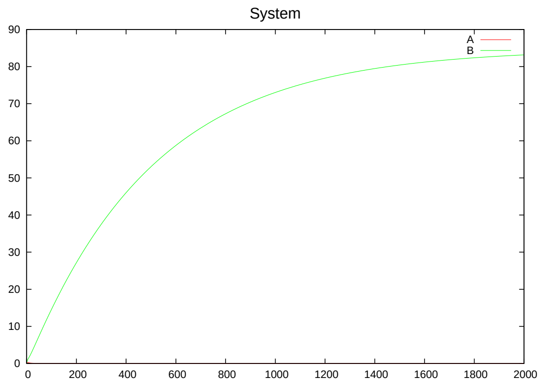
B.8.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.8: Results from the Bistable (in CGF) model.

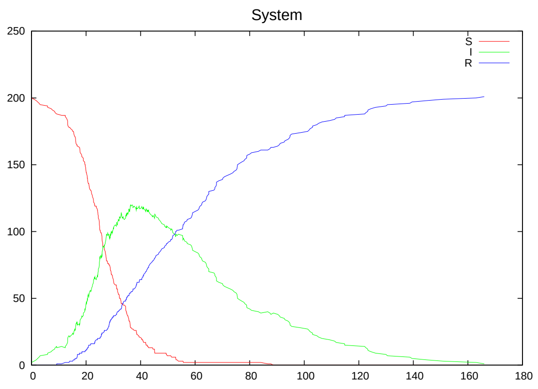
B.9 SIR model

A model adapted from [7].

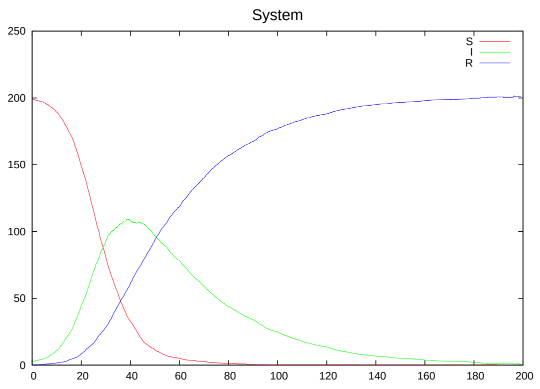
B.9.1 Model

```
var t=0.001;
var r=0.03;

new i@t;
new r@r;

S=?i.I;
I=!i.I + delay@r.R;
R=!r;

System = #200 S|#2 I;
System10 = #2000 S|#20 I;

simulate(System,200.0,1,S,I,R,"output/SIRSim");
odesolve(System,200.0,0.1,10,S,I,R,"output/SIRODE");
rsimulate(System,200.0,20,0.1,1,S,I,R,"output/SIRRSim");


```

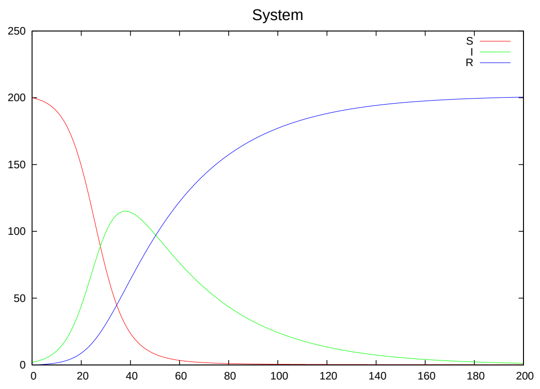
B.9.2 Results



(a) simulation



(b) replicated simulation



(c) ODE solution

Figure B.9: Results from the SIR model.

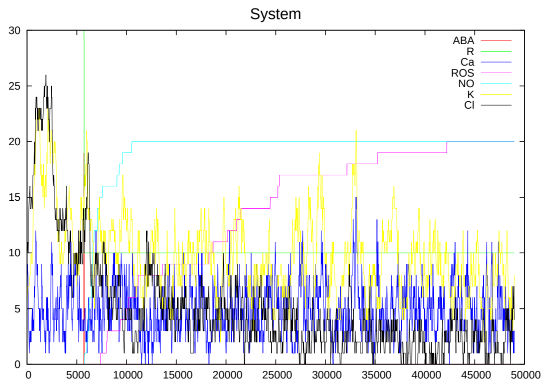
B.10 ABA Signal Transduction in Plants

Taken from [19].

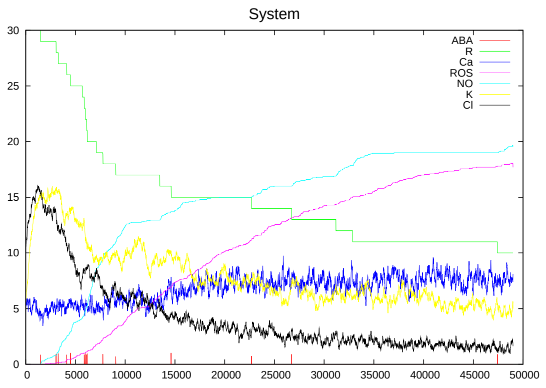
B.10.1 Model

```
 var  abar_delay = 2.0;
 var  abar_to_ros = 0.001;
 var  ros_to_ca = 0.0001;
 var  ca_to_k = 0.01;
 var ca_add = 0.05;
 var ca_decay = 0.000001;
 var k_add   = 0.02;
 var k_decay = 0.001;
 var cl_add = 0.02;
 var cl_decay = 0.001;

 new ababind@1.0;
 new cain@1.0;
 new caout@1.0;
 new kin@0.001;
 new kout@0.00001;
 new kescape@10.0;
 new clin@0.001;
 new clout@0.00001;
 new clescape@10.0;
 new ros@1.0;
 new no@1.0;

 ABA_Helper = delay@0.0001.(#20 ABA);
 ABA = !ababind;
 R = ?ababind.delay@abar_delay.ABAR;
 ABAR = delay@abar_to_ros.(#1 ROS_Intermediate|#1 NO);
 ROS_Intermediate = delay@ros_to_ca.(#2 !cain | ROS);
 ROS = !ros;
 NO = !no;
 Ca = delay@ca_to_k.(Ca_to_K|Ca_to_Cl) +delay@ca_decay;
 Ca_Channel1 = delay@ca_add.(#1 Ca |Ca_Channel1 | Ca_Channel2);
 Ca_Channel2 = ?cain.?cain.(#1 Ca |Ca_Channel1 | Ca_Channel2);
 Ca_to_K = delay@0.001.(#1 !kescape );
 Ca_to_Cl = delay@0.001.(#1 !clout);
 K  = ?kout.?kout + delay@k_decay;
 K_Channel = delay@k_add.(#1 K|K_Channel);
 K_Escape = ?kescape.?kescape.(!kout |K_Escape);
 Cl = ?clout.?clout + delay@cl_decay;
 Cl_Channel = delay@cl_add.(#1 Cl|Cl_Channel);
 Cl_Escape = ?clescape.?clescape.(!clout | Cl_Escape);

 System = (#1 ABA_Helper| #30 R| #1 Ca_Channel1 | 1# Ca_Channel2 |
       #1 K_Channel| #1 Cl_Channel| #5 Ca| #5 K | #10 Cl| #1 K_Escape );
 System10 = (#10 ABA_Helper| #300 R| #10 Ca_Channel1 |#10 Ca_Channel2 |
       #10 K_Channel| #10 Cl_Channel| #50 Ca|#50 K | #100 Cl| #10 K_Escape );
 System100 = (#100 ABA_Helper| #3000 R| #100 Ca_Channel1 | #100 Ca_Channel2 |
       #100 K_Channel| #100 Cl_Channel| #500 Ca| #500 K | #1000 Cl| #100 K_Escape );

simulate(System,49000.0,7,ABA,R,Ca,ROS,NO,K,Cl,"");
rsimulate(System,49000.0,20,10.0,7,ABA,R,Ca,ROS,NO,K,Cl,"");
odesolve(System,49000.0,10.0,1000,ABA,R,Ca,ROS,NO,K,Cl,"");
```

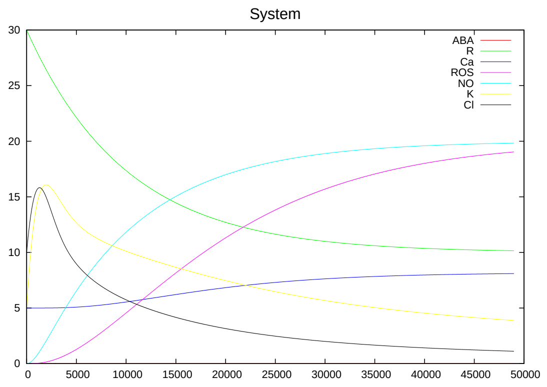
B.10.2 Results



(a) simulation



(b) replicated simulation



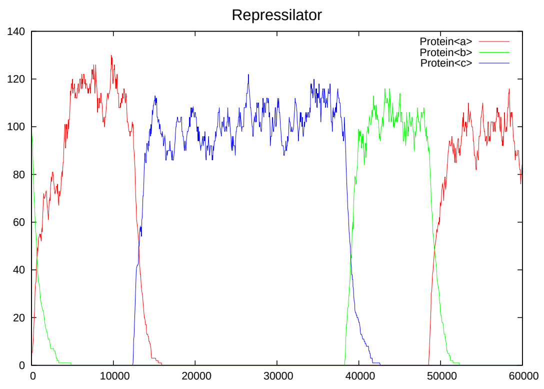
(c) ODE solution

Figure B.10: Results from the ABA signal transduction in plants model.

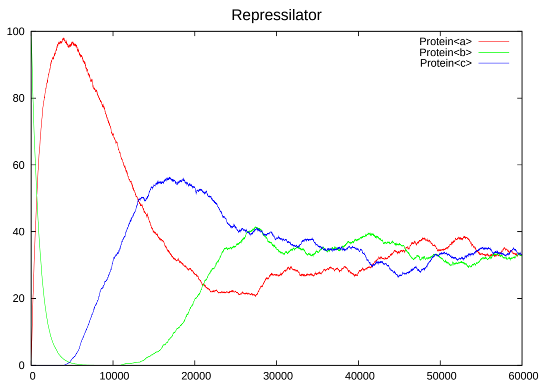
B.11 Repressilator

Translated from the SPiM library[30].

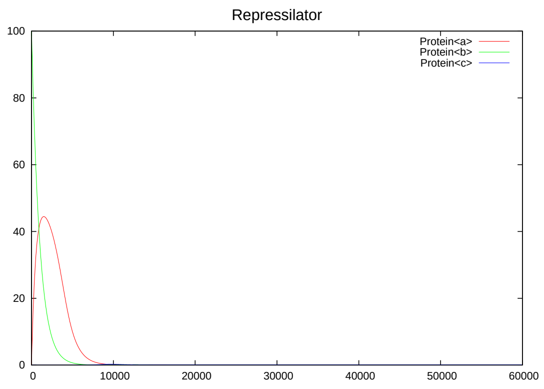
```
//each action has to have a corresponding new defined
var t = 0.1;
var d = 0.001;
var u = 0.0001;

new a@ 1.0;
new b@ 1.0;
new c@ 1.0;


Gene(a,b) = delay@t.(Protein<b> | Gene<a,b>) + ?a.delay@u.Gene<a,b>;
Protein(b) = !b.Protein<b> + delay@d;

Repressilator =  ( Gene<a,b> | Gene<b,c> | Gene<c,a>);

RepressilatorKnocked = (delay@u.Gene<a,b>| #100 Protein<a> |
                        delay@u.Gene<b,c> | delay@u.Gene<c,a>);


//odesolve(RepressilatorKnocked,60000.0,20.0,1000,Protein<a>,Protein<b>,Protein<c>,"");
//rsimulate(RepressilatorKnocked,60000.0,100,10.0,1,Protein<a>,Protein<b>,Protein<c>,"");
simulate(RepressilatorKnocked,60000.0,10,Protein<a>,Protein<b>,Protein<c>,"");
//simulate(Repressilator,600000.0,"",40,Protein<a>,Protein<b>,Protein<c>);
```



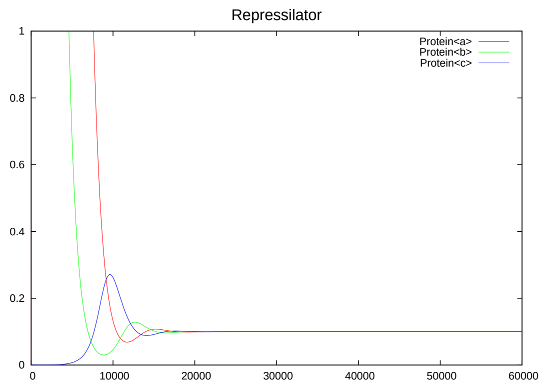
(a) simulation



(b) replicated simulation



(c) ODE solution



(d) detail of ODE solution

Figure B.11:

B.12 Plant Tissue in $\mathcal{L}\pi$

Original model from Chapter 6.

```
new attack@10.0;
new warn@10.0;
new defeat@1.0;
new fight@20.0;
new defeated@10.0;

Cell = ?attack.(L|#6 Resistance| #4 Warning) + ?warn.RCell;
RCell = ?attack.(L|#20 Resistance| #4 Warning) + ?warn.RCell;
Virus = !attack.(!fight.(#2 Virus) + ?defeat);
Resistance = !defeat.!defeated;
L = ?fight+?defeated.Cell;
Warning = !warn;

spatial Tissue = {
        [
                a00: Cell, a01: Cell, a02: Cell, a03: Cell,
                a10: Cell, a11: Cell|Virus, a12: Cell, a13: Cell,
                a20: Cell, a21: Cell, a22: Cell, a23: Cell,
                a30: Cell, a31: Cell, a32: Cell, a33: Cell
        ];
        //row 0
        m(a00,a01,Warning)=10.0;
        m(a00,a10,Warning)=10.0;

        m(a01,a00,Warning)=10.0;
        m(a01,a02,Warning)=10.0;
        m(a01,a11,Warning)=10.0;

        m(a02,a01,Warning)=10.0;
        m(a02,a03,Warning)=10.0;
        m(a02,a12,Warning)=10.0;

        m(a03,a01,Warning)=10.0;
        m(a03,a13,Warning)=10.0;
        //row 1
        m(a10,a11,Warning)=10.0;
        m(a10,a00,Warning)=10.0;
        m(a10,a20,Warning)=10.0;

        m(a11,a10,Warning)=10.0;
        m(a11,a12,Warning)=10.0;
        m(a11,a01,Warning)=10.0;
        m(a11,a21,Warning)=10.0;

        m(a12,a11,Warning)=10.0;
        m(a12,a13,Warning)=10.0;
        m(a12,a02,Warning)=10.0;
        m(a12,a22,Warning)=10.0;

        m(a13,a12,Warning)=10.0;
        m(a13,a03,Warning)=10.0;
        m(a13,a23,Warning)=10.0;
        //row 2
        m(a20,a21,Warning)=10.0;
        m(a20,a10,Warning)=10.0;
        m(a20,a30,Warning)=10.0;

        m(a21,a20,Warning)=10.0;
        m(a21,a22,Warning)=10.0;
        m(a21,a11,Warning)=10.0;
        m(a21,a31,Warning)=10.0;

        m(a22,a21,Warning)=10.0;
        m(a22,a23,Warning)=10.0;
        m(a22,a12,Warning)=10.0;
        m(a22,a32,Warning)=10.0;

        m(a23,a22,Warning)=10.0;
        m(a23,a13,Warning)=10.0;
        m(a23,a33,Warning)=10.0;
        //row 3
        m(a30,a31,Warning)=10.0;
        m(a30,a20,Warning)=10.0;

        m(a31,a30,Warning)=10.0;
        m(a31,a32,Warning)=10.0;
        m(a31,a21,Warning)=10.0;

        m(a32,a31,Warning)=10.0;
        m(a32,a33,Warning)=10.0;
        m(a32,a22,Warning)=10.0;

        m(a33,a32,Warning)=10.0;
        m(a33,a23,Warning)=10.0;


        m(a00,a01,Virus)=0.5;
        m(a00,a10,Virus)=0.5;

        m(a01,a00,Virus)=0.5;
        m(a01,a02,Virus)=0.5;
        m(a01,a11,Virus)=0.5;

        m(a02,a01,Virus)=0.5;
        m(a02,a03,Virus)=0.5;
        m(a02,a12,Virus)=0.5;

        m(a03,a01,Virus)=0.5;
        m(a03,a13,Virus)=0.5;
        //row 1
        m(a10,a11,Virus)=0.5;
        m(a10,a00,Virus)=0.5;
        m(a10,a20,Virus)=0.5;

        m(a11,a10,Virus)=0.5;
        m(a11,a12,Virus)=0.5;
        m(a11,a01,Virus)=0.5;
        m(a11,a21,Virus)=0.5;

        m(a12,a11,Virus)=0.5;
        m(a12,a13,Virus)=0.5;
        m(a12,a02,Virus)=0.5;
        m(a12,a22,Virus)=0.5;

        m(a13,a12,Virus)=0.5;
        m(a13,a03,Virus)=0.5;
        m(a13,a23,Virus)=0.5;
        //row 2
        m(a20,a21,Virus)=0.5;
        m(a20,a10,Virus)=0.5;
        m(a20,a30,Virus)=0.5;

        m(a21,a20,Virus)=0.5;
        m(a21,a22,Virus)=0.5;
        m(a21,a11,Virus)=0.5;
        m(a21,a31,Virus)=0.5;

        m(a22,a21,Virus)=0.5;
        m(a22,a23,Virus)=0.5;
        m(a22,a12,Virus)=0.5;
        m(a22,a32,Virus)=0.5;

        m(a23,a22,Virus)=0.5;
        m(a23,a13,Virus)=0.5;
        m(a23,a33,Virus)=0.5;
        //row 3
        m(a30,a31,Virus)=0.5;
        m(a30,a20,Virus)=0.5;

        m(a31,a30,Virus)=0.5;
        m(a31,a32,Virus)=0.5;
        m(a31,a21,Virus)=0.5;

        m(a32,a31,Virus)=0.5;
        m(a32,a33,Virus)=0.5;
        m(a32,a22,Virus)=0.5;

        m(a33,a32,Virus)=0.5;
        m(a33,a23,Virus)=0.5;
};

spatialSimulate(Tissue,20.0,100,Cell,RCell,Virus,"");
```

C Tests

The different tests used to test the basic syntax and structural congruence of JSPiM.

```
new x@1.0;
new a@2.0;
new b@1.0;

P(x) = !x;
Q(x,y) = !x<y>;

R = !x.R;
S = ?x.S;

T = (new x@2.0)(!x);

//necessary congruences:
        //zero tests
                0 ~ 0;
        //action tests
                !a<x> ~ !a<x>;
                !a<x,y> ~ !a<x,y>;
                !a<x> !~ !a<y>;
                !a<x,y> !~ !a<b,c>;
                !a<x> !~ !b<x>;
                !a<x> !~ ?a(x);
                ?a(x) !~ ?a(y); //alpha congruence not implemented
        //restriction tests
                (new x@2.0)(0) ~ (new x@2.0)(0);
                (new x@2.0 y@2.0)(0) ~ (new x@2.0 y@2.0)(0);
                !a<x> !~ (new x@2.0)(!a<x>);
                (new x@2.0)((new y@2.0)(0)) ~ (new x@2.0 y@2.0)(0);
                (new x@2.0)(!x)|(new x@2.0)(!x)|!x ~ (new x0@2.0 x1@2.0)(!x0|!x1|!x);
                #4 (new x@2.0)(!x)|!x ~
                        (new x0@2.0 x1@2.0 x2@2.0 x3@2.0)(!x0|!x1|!x2|!x3|!x);
        //summation tests
                !a<x> + !b<x> ~ !a<x> + !b<x>;
        //process id tests
                P<a> ~ P<a>;
                P<a> !~ P<b>;
                Q<a,b> ~ Q<a,b>;
                Q<a,b> !~ Q<b,a>;
        //parallel tests
                P<a>|P<b> ~ P<b>|P<a>;
                P<a>|P<a>|P<a>|P<b> ~ P<a>|P<b>|P<a>|P<a>;
                #23 P<a> ~ #23 P<a>;
                #20 P<a>|#21 P<b>|#22 P<c> ~ #21 P<b> | #22 P<c> | #20 P<a>;
                #20 P<a> ~ #10 P<a> | #10 P<a>;
                #1 P<a> !~ #2 P<a>;
                #20 (P<a> | P<b>) ~ #20 P<a> | #20 P<b>;
        //parallel+restriction
                (new a@2.0)(P<a>)|(new b@2.0)(P<b>) ~ (new a@2.0 b@2.0)(P<a>|P<b>);

//optional congruences
//P<a> ~ !a;
!a<x> + !b<x> ~ !b<x> + !a<x>;

//reductions
        ?x.P<a>|!x.P<b> -> P<a>|P<b>;
        ?x(x).P<x>|!x<a> -> P<a>;
        #20 P<a> | ?a -> #19 P<a>;
        //this doesn't work now
        //(new a@2.0)(!x<a>)|?x(x).P<x> -> (new a@2.0)(!a);

        (new a@2.0)(!x<a>)|?x(x).P<x> -> (new a@2.0)(P<a>);
        (new a@2.0)(!x<a>)|?x(x).Q<a,x> -> (new x0@2.0)(Q<a,x0>);
        (!x<a>+!x<b>)|?x(y).P<y> -> P<a>;
        (!x<a>+!x<b>)|?x(y).P<y> -> P<b>;
        R|S -> R|S;

//no reductions
         0 -|;
         !a|!b -|;
         (new x@2.0)(!x)|?x -|;
         #500 !a -|;
         R -|;
```