A Correct Abstract Machine for the Stochastic Pi-calculus

Andrew Phillips [1] Luca Cardelli [2]

Microsoft Research
7 JJ Thomson Avenue
Cambridge, UK

Abstract

In this paper, an abstract machine is presented for a variant of the stochastic pi-calculus, in order to correctly model the stochastic simulation of biological processes. The machine is first proved sound and complete with respect to the calculus, and then used as the basis for implementing a stochastic simulator. The correctness of the stochastic machine helps ensure that the simulator is correctly implemented, giving greater confidence in the simulation results. A graphical representation for the pi-calculus is also introduced.

Key words: abstract machine, stochastic, pi-calculus, correctness, implementation, graphical.

1 Introduction

Process calculi have been seen traditionally as a theoretical framework for the study of concurrent computation, or as a paradigm for more practical concurrent languages, or as a specification language for software and hardware systems that are coded in more pragmatic ways. Therefore, the direct implementation of process calculi for the purpose of execution has never been a high-priority enterprise. Recently, though, a range of process calculi have been adapted or freshly developed for applications in biology, where highly concurrent processes are the norm. In this application domain, process calculi do not act as a paradigm, but as a direct way to describe systems. Therefore, there is a new interest in correct implementation techniques for process calculi, particularly if a quantitative aspect can be added for the purpose of stochastic execution. This paper focuses on implementation techniques for a

[1] Email: anp@imperial.ac.uk
[2] Email: luca@microsoft.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

variant of the stochastic pi-calculus, in order to correctly model the stochastic simulation of biological processes.

The remainder of the paper is structured as follows. In Section 2 a variant of the stochastic pi-calculus is described, along with a corresponding graphical representation. In Section 3 an abstract machine for the stochastic pi-calculus is presented, and in Section 4 the machine is proved sound and complete with respect to the calculus. An implementation of the stochastic machine is described in Section 5, and preliminary simulation results are reported.

2 The Stochastic Pi-Calculus

The variant of the stochastic pi-calculus used in this paper is summarised in Definitions 2.1, 2.2 and 2.3. The calculus is based largely on [5] and [1], but uses a form of guarded replication presented in [7], which simplifies the implementation. Each channel x is associated with a corresponding reaction rate given by $rate(x)$, and each reduction is labelled with the corresponding rate as in [1].

$P, Q ::= \nu x\, P$	Restriction		$\Sigma ::= \mathbf{0}$	Null
$\mid P \mid Q$	Parallel		$\mid \pi.P + \Sigma$	Action
$\mid \Sigma$	Summation		$\pi ::= x\langle n\rangle$	Output
$\mid\ !\pi.P$	Replication		$\mid x(m)$	Input

Definition 2.1 *Syntax of* SPi

(1) $Q \equiv P \wedge P \xrightarrow{r} P' \wedge P' \equiv Q' \quad \Rightarrow \quad Q \xrightarrow{r} Q'$

(2) $P \xrightarrow{r} P' \quad \Rightarrow \quad \nu x\, P \xrightarrow{r} \nu x\, P'$

(3) $P \xrightarrow{r} P' \quad \Rightarrow \quad P \mid Q \xrightarrow{r} P' \mid Q$

(4) $(x\langle n\rangle.P + \Sigma) \mid (x(m).Q + \Sigma') \xrightarrow{rate(x)} P \mid Q_{\{n/m\}}$

Definition 2.2 *Reduction in* SPi

A graphical syntax for the pi-calculus is presented in Figure 1, which corresponds to the written syntax of Definition 2.1. According to Figure 1, a graphical process is a tree of nodes, where a Restriction node consists of a dotted line linking the restricted name to the corresponding process, and a Parallel node consists of two adjacent processes enclosed in a box. A Summation node is a null node with zero or more labelled arcs, and a Replication node is a bang node with a single labelled arc. Each arc connects to a process, and is labelled with an input or output action. For convenience, *links* between nodes in the tree can be encoded in order to represent recursive processes,

(5) $P \equiv_\alpha Q \quad \Rightarrow \quad P \equiv Q$

(6) $P \mid \mathbf{0} \equiv P$

(7) $P \mid Q \equiv Q \mid P$

(8) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

(9) $!\pi.P \equiv \pi.(P \mid !\pi.P) + \mathbf{0}$

(10) $x \notin fn(P) \quad \Rightarrow \quad \nu x\,(P \mid Q) \equiv P \mid \nu x\,Q$

(11) $\nu x\,\mathbf{0} \equiv \mathbf{0}$

(12) $\nu x\,\nu y\,P \equiv \nu y\,\nu x\,P$

(13) $\pi.P + \pi'.P' + \Sigma \equiv \pi'.P' + \pi.P + \Sigma$

(14) $\Sigma \equiv \Sigma' \quad \Rightarrow \quad \pi.P + \Sigma \equiv \pi.P + \Sigma'$

Definition 2.3 *Structural congruence in* SPi

where a link is a labelled arc with a double-headed arrow. Each node P with an inbound link is encoded as $\nu x\,(!x().P \mid x\langle\rangle)$, and each link to this node with label π is encoded as $\pi.x\langle\rangle$, where the scope of x is extended accordingly.

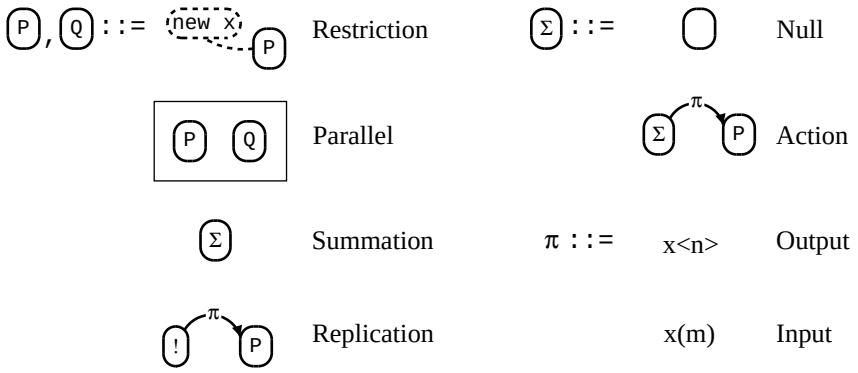

Figure 1. Graphical SPi Syntax

The graphical syntax can be used to model the regulation of gene expression by positive feedback based on [9], as shown in Figure 2, where each parallel process describes the behaviour of a different molecule or entity in the system. For clarity, certain nodes in the figure are annotated with names, but the names themselves do not have any semantic meaning. According to Figure 2, when the Replication node $!Protein\_A$ receives an input on channel $protein\_A$, a new protein A is spawned in parallel. Graphically, this is represented by spawning a parallel copy of the graph which follows the input on $protein\_A$. Each protein A can bind with a protein TF by sending private *unbind*, *send*, and *remove* channels on the *bind* channel. Once bound, A can send a protein *tail* to TF on the *send* channel. After TF unbinds from A, it can use its newly acquired *tail* to increase transcription promotion of DNA_A or DNA_TF, which are subsequently transcribed into RNA_A or RNA_TF at a higher rate, which in turn can be translated into A or TF

proteins, resulting in a positive feedback loop. A single *Proteins* process is used to represent the basic functions of degradation, transcription and translation performed by other proteins in the organism. Additional details can be found at [6].

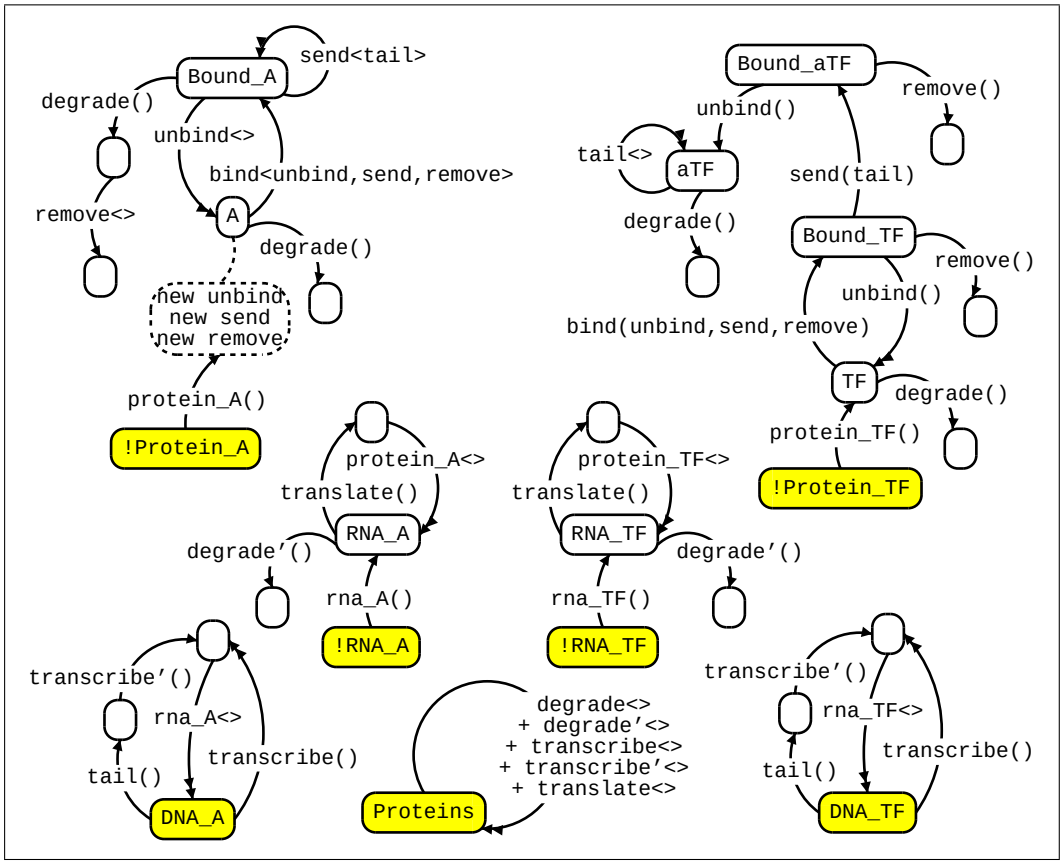

Figure 2. Regulating Gene Expression by Positive Feedback [9]

3 The Stochastic Pi-Machine

3.1 Approach

The Stochastic Pi-Machine (SPiM) is a formal description of how a stochastic pi-calculus process can be executed. The machine is inspired by recent work on abstract machines for process calculi [7,12] and uses a list syntax, which is close to an implementation language. The SPi-Machine executes a given process P by first encoding P into a list of summations with a number of top-level private names. The machine then uses a stochastic selection algorithm based on [2] to choose a particular channel x on which to perform a communication. The procedure is repeated until no more communications are possible. A detailed description of the SPi-Machine is given in the remainder of this section.

3.2 Encoding

In order to execute a given process P, the SPi-Machine first needs to encode P into a suitable machine term. The set of machine terms is denoted by SPiM, and individual machine terms V, U are defined using lists A, B. According to Definition 3.1, a machine term V is a list with zero or more restricted names, and a list A is either an empty list $[]$ or a list containing one or more summations Σ. Note that summations in SPiM are identical to summations in SPi.

$$\begin{array}{rcll} V, U & ::= & \nu x\, V & \text{Restriction} \\ & | & A & \text{List} \\ A, B & ::= & [] & \text{Empty} \\ & | & \Sigma :: A & \text{Summation} \end{array}$$

Definition 3.1 *Syntax of* SPiM

The SPi-Machine encodes a given process P into a machine term using an *encoding function* $(\!|P|\!)$. According to Definition 3.2, a process P is encoded by adding it to an empty list $[]$ using a *construction operator* $\circ$.

$$(\!|P|\!) \triangleq P \circ []$$

Definition 3.2 *Encoding*

The construction operator $P \circ V$ adds a process P to an arbitrary machine term V, producing an updated machine term as a result. According to Definition 3.3, if a process P is added to a term $\nu x\, V$ containing a private name x, then P is added to V and the scope of x is extended to the top level, provided x is not known to P (15). Once the scope of each private name has been extended in this way, the process P can be added to the remaining list A. The null process $\mathbf{0}$ is not added to the list (16), and the parallel composition process $P \mid Q$ is split so that each parallel process is added separately (17). The restriction process $\nu y\, P$ is modified by replacing y with a fresh name x, the scope of x is extended to the top level and the process $P_{\{x/y\}}$ is added to the list (18). The replicated action $!\pi.P$ is expanded to a summation consisting of a single action, and the resulting summation is added to the list (19). Finally, the non-empty summation $\pi.P + \mathbf{0}$ is placed at the head of the list (20).

3.3 Execution

Once a process has been encoded to a machine term using the construction operator, it can then be executed by the machine. In general, a machine term

$$n \notin fn(P) \quad \Rightarrow \quad P \circ (\nu x\, V) \triangleq \nu x\, (P \circ V) \tag{15}$$

$$\mathbf{0} \circ A \triangleq A \tag{16}$$

$$(P \mid Q) \circ A \triangleq P \circ Q \circ A \tag{17}$$

$$x \notin fn(P \circ A) \quad \Rightarrow \quad (\nu y\, P) \circ A \triangleq \nu x\, (P_{\{x/y\}} \circ A) \tag{18}$$

$$!\pi.P \circ A \triangleq (\pi.(P \mid !\pi.P) + \mathbf{0}) \circ A \tag{19}$$

$$(\pi.P + \Sigma) \circ A \triangleq (\pi.P + \Sigma) :: A \tag{20}$$

Definition 3.3 *Construction in* SPiM

is a list of summations with a number of top-level private names:

$$\nu x_1\, \nu x_2\, ...\nu x_N\, (\Sigma_1 :: \Sigma_2 :: ... :: \Sigma_M :: [])$$

A given term is executed by the machine in steps, according to a labelled reduction relation $\xrightarrow{r}$. The relation $V \xrightarrow{r} V'$ is true if the machine can transform a term V into a term V' with rate r during a single execution step. According to Definition 3.4, if a term V can reduce to V' with rate r then this reduction can also take place if V contains a private name x (21). This rule allows the machine to execute a list A with an arbitrary number of private names. The machine executes a list A by first choosing the next channel x on which to perform a communication, using the function $Next(A)$. The machine then uses a *selection operator* $\succ$ to choose a summation $x(m).P + \Sigma$ with an input on channel x and another summation $x\langle n\rangle.Q + \Sigma'$ with an output on x. The value n is then sent along channel x and bound to m in process P. The summations Σ and Σ' are discarded, and the processes $P_{\{n/m\}}$ and Q are added to the remainder of the list (22).

$$V \xrightarrow{r} V' \quad \Rightarrow \quad \nu x\, V \xrightarrow{r} \nu x\, V' \tag{21}$$

$$\left.\begin{array}{l} x = Next(A) \\ \wedge A \succ (x(m).P + \Sigma) :: A' \\ \wedge A' \succ (x\langle n\rangle.Q + \Sigma') :: A'' \end{array}\right. \quad \Rightarrow \quad A \xrightarrow{rate(x)} P_{\{n/m\}} \circ Q \circ A'' \tag{22}$$

Definition 3.4 *Reduction in* SPiM

The selection operator $\succ$ chooses a particular action from inside a list by first moving a summation to the head of the list and then moving an action to the front of the summation. The relation $A \succ B$ is true if the list A can be re-arranged to match list B. According to Definition 3.5, a list can match itself (23) or it can be re-arranged by bringing one of its summations to the head of the the list (24). Finally, a summation at the head of a list can be re-arranged by bringing one of its actions to the front of the summation (25). Note that, for efficiency reasons, the selection operator only allows a single

action inside a single summation to be selected, leaving the remainder of the list unaltered. This prevents the contents of the list from being permuted arbitrarily.

(23) $$A \succ A$$

(24) $$A \succ \Sigma' :: A' \Rightarrow \Sigma :: A \succ \Sigma' :: \Sigma :: A'$$

(25) $$\Sigma :: A \succ (\pi'.P' + \Sigma') :: A \Rightarrow (\pi.P + \Sigma) :: A \succ (\pi'.P' + \pi.P + \Sigma') :: A$$

Definition 3.5 *Selection in* SPiM

The next reaction channel x and the reaction delay τ are calculated using the algorithm described in Definition 3.6. The algorithm is based on the Gillespie algorithm [2], which uses a notion of *channel activity* in order to stochastically select the next reaction channel. A similar notion of channel activity is defined for the SPi-Machine, where $\mathrm{Act}_x(A)$ denotes the activity of channel x in list A. The activity corresponds to the number of possible combinations of inputs and outputs on channel x in A, and is defined by:

$$\mathrm{Act}_x(A) = (\mathrm{In}_x(A) * \mathrm{Out}_x(A)) - \mathrm{Mix}_x(A)$$

where $\mathrm{In}_x(A)$ and $\mathrm{Out}_x(A)$ are the number of unguarded *inputs* and *outputs* on channel x in A, respectively, and $\mathrm{Mix}_x(A)$ = the sum of $\mathrm{In}_x(\Sigma_i) \times \mathrm{Out}_x(\Sigma_i)$ for each summation Σ_i in A. The formula takes into account the fact that an input and an output in the same summation cannot interact, by subtracting $\mathrm{Mix}_x(A)$ from the product of the number of inputs and outputs on x. Once the values x and τ have been calculated, the machine increments the reaction time by delay τ and randomly chooses one of the available reactions on x with equal probability, using the selection operator. This is achieved by randomly choosing a number $n \in [1..\mathrm{In}_x(A)]$ and selecting the nth input in A, followed by randomly selecting an output from the remaining list in a similar fashion.

(i) For all $x \in fn(A)$ calculate $a_x = \mathrm{Act}_x(A) * rate(x)$

(ii) Store non-zero values of a_x in a list (x_μ, a_μ), where $\mu \in 1...M$.

(iii) Calculate $a_0 = \sum_{\nu=0}^{M} a_\nu$

(iv) Generate two random numbers $n_{1,} n_2 \in [0, 1]$ and calculate τ, μ such that:

$$\tau = (1/a_0)\ln(1/n_1)$$

$$\sum_{\nu=1}^{\mu-1} a_\nu < n_2 a_0 \leq \sum_{\nu=1}^{\mu} a_\nu$$

(v) $Next(A) = x_\mu$ and $Delay(A) = \tau$.

Definition 3.6 *Calculating* $Next(A)$ *and* $Delay(A)$ *according to [2]*

For improved efficiency, the machine can store a list of tuples for each channel x in A, of the form:

$$x, \mathrm{In}_x(A), \mathrm{Out}_x(A), \mathrm{Mix}_x(A), a_x$$

After each reduction has been performed, it is only necessary to update the values for those channels that were affected by the reduction, and then use Definition 3.6 on the updated values to choose the next reaction channel and calculate the delay.

4 Correctness of the Stochastic Pi-Machine

4.1 Approach

The correctness of the SPi-Machine is expressed in terms of five main properties: *safety*, *soundness*, *completeness*, *termination* and *duration*. Safety ensures that the machine does not produce any runtime errors, and Soundness ensures that the machine can only perform valid execution steps. Completeness is a much stronger property, which ensures that the machine can accurately execute all possible behaviours of the calculus. Termination ensures that the machine does not loop forever unnecessarily, and Duration ensures that each reduction in the machine takes the same length of time as the corresponding reduction in the calculus, and vice-versa. The details of the proofs can be found at [6].

4.2 Safety

Safety ensures that the machine does not produce any runtime errors when executing a given term V. According to Lemma 4.1, if the machine reduces a term V to V' with rate r, then V' will be a valid machine term.

Lemma 4.1 $\forall V.V \in \mathrm{SPiM} \wedge V \xrightarrow{r} V' \Rightarrow V' \in \mathrm{SPiM}$

Proof By Lemma 4.2, Lemma 4.3 and by induction on Definition 3.4 of reduction in SPiM. □

Lemma 4.2 $\forall A \in \mathrm{SPiM}.A \succ B \Rightarrow B \in \mathrm{SPiM}$

Proof By induction on Definition 3.5 of selection in SPiM. □

Lemma 4.3 $\forall V.\forall P.V \in \mathrm{SPiM} \wedge P \in \mathrm{SPi} \Rightarrow P \circ V \in \mathrm{SPiM}$

Proof By induction on Definition 3.3 of construction in SPiM. □

4.3 Soundness

Soundness ensures that each reduction in the machine corresponds to a valid reduction in the calculus. In order to prove the soundness of the machine it is necessary to define a *decoding function* $[\![V]\!]$, which maps a given machine term V to a corresponding calculus process. According to Definition 4.4, a term

$\nu x\, V$ with a private name x is mapped to the decoded term $[\![V]\!]$ with a private name x (26). The null list is mapped to the null process (27), and a summation at the head of a list is mapped to a summation in parallel with the decoded list (28). Lemma 4.5 ensures that the decoding function is well-defined.

(26) $$[\![\nu x\, V]\!] \triangleq \nu x\, [\![V]\!]$$

(27) $$[\![[]]\!] \triangleq \mathbf{0}$$

(28) $$[\![\Sigma :: A]\!] \triangleq \Sigma \mid [\![A]\!]$$

Definition 4.4 *Decoding*

Lemma 4.5 $\forall V.V \in \text{SPiM} \Rightarrow [\![V]\!] \in \text{SPi}$

Proof By induction on Definition 4.4 of decoding in SPiM. □

Once a decoding from machine terms to calculus processes has been defined in this way, it is possible to state and prove the soundness of the machine. According to Theorem 4.6, if the machine can reduce a term V to V' with rate r, then the calculus can perform a corresponding reduction with the same rate on the decoding of V.

Theorem 4.6 $\forall V.V \in \text{SPiM} \wedge V \xrightarrow{r} V' \Rightarrow [\![V]\!] \xrightarrow{r} [\![V']\!]$

Proof By Lemma 4.7, Lemma 4.8 and by induction on Definition 3.4 of reduction in SPiM. □

Lemma 4.7 $\forall A.A \in \text{SPiM} \wedge A \succ B \Rightarrow [\![A]\!] \equiv [\![B]\!]$

Proof By induction on Definition 3.5 of selection in SPiM. □

Lemma 4.8 $\forall V.\forall P.V \in \text{SPiM} \wedge P \in \text{SPi} \Rightarrow [\![P \circ V]\!] \equiv P \mid [\![V]\!]$

Proof By induction on Definition 3.3 of construction in SPiM. □

4.4 Completeness

Completeness ensures that each reduction in the calculus can be matched by a corresponding reduction in the machine, up to re-ordering of machine terms. In order to prove the completeness of the machine it is necessary to define a structural congruence relation $V \equiv U$, which allows a term V to be re-ordered to match a term U. According to Definition 4.9, terms are structurally congruent up to alpha-conversion (29), unused private names can be discarded (30), private names can be permuted (31), summations inside a list can be permuted (32)-(33) and actions inside a summation can also be permuted (34)-(35).

An important property of structural congruence is that congruent terms should be able to perform corresponding reductions that preserve the congruence relation. This property needs to be proved explicitly for the machine,

$$V \equiv_\alpha U \Rightarrow V \equiv U \tag{29}$$

$$x \notin fn(V) \quad \Rightarrow \quad \nu x\, V \equiv V \tag{30}$$

$$\nu x\, \nu y\, V \equiv \nu y\, \nu x\, V \tag{31}$$

$$\Sigma :: \Sigma' :: A \equiv \Sigma' :: \Sigma :: A \tag{32}$$

$$A \equiv A' \quad \Rightarrow \quad \Sigma :: A \equiv \Sigma :: A' \tag{33}$$

$$(\pi.P + \pi'.P' + \Sigma) :: A \equiv (\pi'.P' + \pi.P + \Sigma) :: A \tag{34}$$

$$\Sigma :: A \equiv \Sigma' :: A \Rightarrow (\pi.P + \Sigma) :: A \equiv (\pi.P + \Sigma') :: A \tag{35}$$

Definition 4.9 *Structural Congruence in* SPiM

since structural congruence is not used in the definition of reduction. The omission is deliberate, and avoids the need to examine all possible re-orderings of a term in order to perform a reduction. As a result, the efficiency of the machine is significantly improved from $\mathcal{O}(!n)$ to $\mathcal{O}(n)$, where n is the number of summations in the machine. According to Lemma 4.10, if the machine can reduce a term V to V' with rate r, then it can reduce any term that is congruent to V to a term that is congruent to V', with the same rate.

Lemma 4.10 $\forall V.V \in \text{SPiM} \wedge U \equiv V \wedge V \xrightarrow{r} V' \Rightarrow \exists U'.U \xrightarrow{r} U' \wedge U' \equiv V'$

Proof By induction on Definition 4.9 of structural congruence in SPiM. □

Once a structural congruence relation has been defined in this way, it is possible to state and prove the completeness of the machine. According to Theorem 4.11, if the calculus can reduce a process P to P' with rate r, then the machine can perform a corresponding reduction with the same rate on the encoding of P, up to structural congruence.

Theorem 4.11 $\forall P.P \in \text{SPi} \wedge P \xrightarrow{r} P' \Rightarrow (\!|P|\!) \xrightarrow{r} \equiv (\!|P'|\!)$.

Proof By Lemma 4.12 and by induction on Definition 2.2 of reduction in SPi, where the rule for parallel composition (3) is expanded over the remaining rules (1), (2), (4). □

Lemma 4.12 $P \equiv Q \Rightarrow (\!|P|\!) \equiv (\!|Q|\!)$

Proof By induction on Definition 2.3 of structural congruence in SPi. □

4.5 Termination

Termination ensures that the machine stops executing if there are no more reductions to be performed. This prevents a given simulation from looping forever unnecessarily. According to Theorem 4.13, if a process P cannot reduce, then the corresponding machine term cannot reduce either.

Theorem 4.13 $\forall P.P \in \text{SPi} \wedge P \not\longrightarrow \quad \Rightarrow \quad (\!|P|\!) \not\longrightarrow$

Proof By Theorem 4.6 and by basic relationships between encoding and decoding. □

4.6 Duration

The Gillespie algorithm has been proved correct as a means of stochastically selecting a reaction channel [2], and soundness and completeness both ensure that the machine performs each reduction $\xrightarrow{r}$ with the correct rate. However, these properties are not sufficient to express the correctness of the stochastic machine, as illustrated by the following example:

$$P_1 \triangleq x\langle n\rangle.P + x\langle n\rangle.P \mid x(m).Q$$
$$P_2 \triangleq x\langle n\rangle.P \mid x(m).Q$$

In this example, both P_1 and P_2 can reduce to the same process $P \mid Q_{\{n/m\}}$, with the same reduction $\xrightarrow{rate(x)}$, yet the reduction is twice as fast in process P_1 as it is in process P_2. This is because two competing actions with exponential distributions of rate r can be viewed as a single action with an exponential distribution of rate $2r$, as explained in [8,1]. In order to distinguish between such processes, it is necessary to take into account the number of possible interactions on a chosen channel x in a list A, i.e. the activity $\mathrm{Act}_x(A)$ of x in A. This can be achieved by defining a corresponding notion of channel activity for calculus processes, and ensuring that the activity is preserved by decoding and encoding, as described in Proposition 4.14 and Proposition 4.15 respectively. This ensures that reactions in the machine have the same duration as reactions in the calculus, and vice-versa.

Proposition 4.14 $\forall V \in \mathrm{SPiM}.\mathrm{Act}_x(V) = \mathrm{Act}_x([\![V]\!])$

Proposition 4.15 $\forall P \in \mathrm{SPi}.\mathrm{Act}_x(P) = \mathrm{Act}_x((\!|P|\!))$

5 Implementation

5.1 Approach

A prototype simulator has been implemented in a functional language (OCaml), based on the abstract machine specification. The simulator consists of a single binary executable, which reads in a source file and simulates reactions for a given duration. The simulation results are stored in a log file as a list of comma-separated values, which can be visualised using third-party software. In addition, a polymorphic type system for channel communication has been implemented based on [12], and a static type-checker accurately reports syntax and type errors before a given source file is executed.

5.2 Data Types

The terms of the machine can be readily implemented as functional datatypes. By definition, a term V is a list of summations with a number of top-level private names $\nu x_1 ... \nu x_N\, A$. In practice, however, the privacy of these top-level names does not need to be implemented explicitly, since each simulator will have its own private address space for storing and manipulating names. Therefore, a machine `term` can be implemented as a list of summations, where a summation is a list of (`action`, `process`) pairs. A `name` is implemented as a (`string`, `float`) pair, where the float corresponds to the reaction rate. The implementation also allows constants and tuples to be sent and received over channels, using `value` and `pattern` data types accordingly.

```
type term = ((action*process) list) list

type process =  Null
              | Parallel of process*process
              | Restriction of name*process
              | Replication of (action*process)
              | Summation of (action*process) list

type action = Input of value*pattern
            | Output of value*value
```

5.3 Encoding

The simulator executes a pi-calculus source file, written in a standard ascii syntax, by first parsing the file to produce a corresponding `process`. The process is then encoded to a `term` using a `cons` function to add the process to an empty list. The `cons` function is a direct implementation of the construction operator $\circ$. In particular, a restriction process `Restriction`(`n`, `p`) is added to a term by a generating a fresh name based on `n`, substituting `n` with this fresh name in `p` using the `bind` function, and adding the resulting process to the list. The function `fresh`(`n` : `name`) uses a naming convention to guarantee that each generated name is globally fresh for the duration of the simulation. This can be achieved by appending a time stamp (or a suitable global counter) to the name using a reserved suffix such as $\sim$. If the generated name is globally fresh then the restriction can be brought to the top level according to Definition 3.3, which means that it does not need to be explicitly represented by the machine.

```
let rec cons (p:process) (l:term) = match p with
    Null -> l
  | Parallel(p,p') -> cons p (cons p' l)
  | Restriction(n,p) -> cons (bind (fresh n) n p) l
  | Replication(a,p') -> [a,Parallel(p',p)]::l
  | Summation(s) -> s::l
```

5.4 Execution

The implementation uses the function `reduce(l : term)` to perform a single execution step on a term `l`. The function `gillespie(l : term)` returns a channel inside the list that is able to communicate, along with the time elapsed. Both of these values are calculated according to the stochastic algorithm in Definition 3.6. The function `select(a : action)(l : term)` randomly chooses an action from inside the list `l` that matches the action `a`. The match is performed based solely on the type of the action and the channel, so arbitrary default constants are used for the input pattern `m0` and output value `v0` in the arguments.

```
let reduce (l:term) =
  let (x:value),(t:float) = gillespie l
  in match select (Input(x,m0)) l with
      Some((Input(x,m),p),s,l) -> (
        match select (Output(x,v0)) l with
            Some((Output(x',v),p'),s',l) ->
              if x==x'
              then Some(t,cons (bind v m p) (cons p' l))
              else None
          | _ -> None
      )
    | _ -> None
```

The simulator repeatedly applies the `reduce` function to the list until no more reductions are possible, at which point the simulation terminates. After each reduction step, the machine logs the time elapsed and the quantity of top-level inputs and outputs on each channel. The results are stored in a file as a comma separated list, which can be visualised using third party software. In order to improve the efficiency of the machine, the `reduce` function can be modified to keep track of the total number of inputs, outputs and mixed sums as described in Section 3.

5.5 Simulation Results

The implementation has been used to simulate the regulation of gene expression by positive feedback [9], described in Section 2. As shown in Figure 3 and in accordance with [9], higher levels of Protein A are observed in the presence of the TF gene and lower levels are observed when the TF gene is disabled.

The implementation has also been used to simulate a wide variety of chemical reactions and biological systems, including enzymatic reactions, a circadian clock, and a model of the cell cycle control in eucaryotes [4]. Details of simulation results are available from [6].

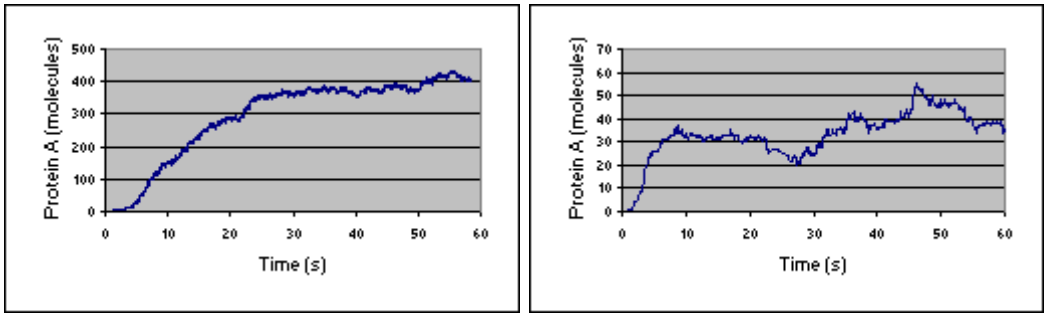

Figure 3. Protein A molecules v.s. time in presence (left) and absence (right) of TF

6 Related Work

The BioSPI system [9] is an existing implementation of a biochemical variant of the stochastic π-calculus. The system executes a process by compiling it to an FCP procedure, which is then executed by the FCP Logix platform [11]. Channel data structures are used to maintain stochastic information and synchronize send and receive requests, in accordance with the Gillespie algorithm. Unlike SPiM, there is no formal definition of an abstract machine, and the implementation is specific to FCP Logix. In addition, according to [9] BioSPI calculates the activity of a channel x in P by $\mathrm{In}_x(P) * \mathrm{Out}_x(P)$. This assumes that there can never be both an input and an output on the same channel in the same summation. A special rate law is defined for homodimerization reactions of the form $\Sigma + x\langle n\rangle.P + x(m).Q \mid \Sigma' + x\langle n\rangle.P' + x(m).Q'$, but this does not account for arbitrary combinations of input and output. Furthermore, due to scope extrusion, it is not clear whether such arbitrary combinations can be avoided statically, without limiting the expressiveness of the calculus. SPiM attempts to address this issue by giving a more general definition of channel activity, which accounts for mixed inputs and outputs using $\mathrm{Mix}_x(P)$. Homodimerization reactions are also included in this definition, provided the reaction rate of the corresponding channel x is halved in the model. It is also worth noting that, unlike SPiM, the current BioSPI system does not implement a type system for channel communication. The BioSPI system has also been extended to handle membrane interactions [10].

Another implementation of the stochastic pi-calculus is the StoPi simulator described in [1], where fully general sums are supported. A stochastic calculus is formally defined, and the implementation architecture is also described in detail, but the paper does not include an abstract machine that maps readily to program code, or prove the correctness of the machine. An alternative stochastic simulator is the PEPA system [3], which can also be used to simulate biological processes. However, PEPA does not include a notion of name-passing, which is important for modelling chemical bonding and is one of the main features of the pi-calculus.

7 Conclusion

We have described an abstract machine for a basic stochastic process calculus, and verified some of its properties. We hope that this will form a framework on which to design and build implementations of richer stochastic process calculi, and possibly of very different, biologically inspired calculi that share a stochastic architecture. We also plan to incorporate a graphical front-end to the current simulator, and generate the corresponding pi-calculus code automatically.

References

[1] Bloch, A., B. Haagensen, M. K. Hoyer and S. U. Knudsen, "The StoPi-calculus and Simulator," Article and runtime system available from http://www.cs.auc.dk/~steffen/dat4/stopi/.

[2] Gillespie, D. T., *Exact stochastic simulation of coupled chemical reactions*, J. Phys. Chem. **81** (1977), pp. 2340–2361.

[3] Gilmore, S. and J. Hillston, *The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling*, in: *Proceedings of MTTCPE*, number 794 in LNCS (1994), pp. 353–368.

[4] Lecca, P. and C. Priami, *Cell cycle control in eukaryotes: a biospi model*, in: *BioConcur'03* (2003).

[5] Milner, R., "Communicating and Mobile Systems: the π-Calculus," 1999.

[6] Phillips, A., "The Stochastic Pi-Machine," Available from http://www.doc.ic.ac.uk/~anp/spim/.

[7] Phillips, A., N. Yoshida and S. Eisenbach, *A distributed abstract machine for boxed ambient calculi*, in: *ESOP'04*, LNCS (2004).

[8] Priami, C., *Stochastic π-calculus*, The Computer Journal **38** (1995), pp. 578–589, proceedings of PAPM'95.

[9] Priami, C., A. Regev, E. Shapiro and W. Silverman, *Application of a stochastic name-passing calculus to representation and simulation of molecular processes*, Information Processing Letters .

[10] Regev, A., E. M. Panina, W. Silverman, L. Cardelli and E. Shapiro, *Bioambients: An abstraction for biological compartments*, in: *Theoretical Computer Science, Special Issue on Computational Methods in Systems Biology*, to Appear.

[11] Silverman, W., M. Hirsch, A. Houri and E. Shapiro, *The logix system user manual, version 1.21*, in: E. Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume II)*, MIT Press, London, 1987 pp. 46–77.

[12] Turner, D. N., "The Polymorphic Pi-Calculus: Theory and Implementation," Ph.D. thesis (1996), cST-126-96 (also published as ECS-LFCS-96-345).