On the universality of variants of P systems

M. Tech Thesis

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Avadhut Sardeshmukh
Roll No: 06329905

under the guidance of

Prof. S N Krishna

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgement

I would like to thank my guide, Prof. S.N.Krishna for her valuable suggestions, guidance and comments during the course of this work.

I take this opportunity to thank Dr. Subhadra for her valuable guidance and support. I am grateful to all my friends including Anil, Jaideep, Vipul, Vivek, Vinayak who were always there to help me. Above all, I would like to thank Raghvendra and Tushar without whose eternal support, I could not have been writing this thesis.

Avadhut Sardeshmukh.

Contents

List of Figures

Abstract

P systems were introduced as a framework for defining distributed, parallel and non-deterministic computing devices inspired by the structure and functioning of living cells. In this work, we study two variants of P systems – namely, P systems using worm objects and spiking neural P systems. The optimality of number of membranes required for a P system with worm objects to be universal is still an open problem. We prove that four membranes suffice. Whether this number is optimal or not still remains an open problem.

The second variant we study is the spiking neural P systems (SN P systems for short). Synchronization plays a central role in proving that these systems are universal. But considering asynchronous systems is of biological as well as mathematical interest, because real biological systems are asynchronous. We want to discover the loss in power, if any, due to loss of synchronization. Asynchronous SN P systems using extended rules (spiking rules that can emit unbounded number of spikes) were proved to be universal, but whether use of extended rules is unavoidable is unclear. We prove that extended rules that can emit up to four spikes are enough to get universality. To compare and contrast the power of synchronous and asynchronous SN P systems, further, we come up with a hierarchy of asynchronous SN P systems with bounded number of neurons (1, 2, 3, etc) and compare the power with corresponding system in synchronous mode.

Chapter 1

Introduction

The area of membrane systems was triggered by a landmark paper by Gheoghe Paun in 1998 in the Turku Center for Computer Science (TUCS) Report. The same paper was later circulated in the Journal of Computer and system sciences in 2000. Till this time natural computing models were *in vitro*. For example, DNA computing. This was the first time that a cell itself was the object of research–viz. membranes in a cell and compartmentalization (*in vivo* structure of the cell).

1.1 Inspiration from Biology

It is well known that the cell is the smallest thing on earth to be unanimously considered as live.The cell has a very exquisite internal structure. It is composed of compartments created by various membranes. An outer membrane defines the cell itself from the environment. The membrane is a semi-permeable barrier between the compartments. Molecules can be transported from one compartment to other via vesicles enclosed by membranes. The membrane structure is hierarchical. The compartments are like "protected reactors" where specific biochemical processes take place.
Our membrane computing model is roughly inspired by these properties of the cell and its internal membrane structure. The main component of this model is the membrane structure. Each membrane of this structure defines a region. Each region has a multiset of objects. And each region has a set of rules (evolution rules or symport/antiport rules) which operate on the multiset of objects in that region. Evolution rules are rewriting rules (like

grammar rules) that represent the biochemical reactions going on in the cell-compartments and symport/antiport rules are transport rules that represent the vesicles through which molecules are transported from one compartment to other. The result of a computation can be the number of objects of a particular kind in a region (called output region), though there are other possibilities, too.

1.2 Formal Definition

We here define the most basic model of membrane systems–those with symbol-objects. Although there exists a large panoply of models of membrane systems, we believe that they can be derived/understood from these basic concepts.

P System with symbol objects:
[Definition] A P system of degree m ≥ 1 with symbol-objects is a tuple :

$$\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0)$$

where

- O is an alphabet and its elements are called objects
- μ is a membrane structure consisting of m membranes arranged in an hierarchical structure; the membranes (and hence the regions they delimit) are labelled with 1,2, ... ,m of μ.
- R_i are finite sets of evolution rules over O; R_i is associated with region i. An evolution rule is of the form $u \rightarrow v$, u is a string over O and v is a string over $\{a_{here}, a_{out} \mid a \epsilon O\} \cup \{a_{in_j} \mid \ a \epsilon O, 1 \leq j \leq m\}$.
- $i_0 \epsilon \{0, 1, 2, \ldots, m\}$ is the output region. In case $i_0 \geq 1$, it is the region enclosed by membrane i_0 and if $i_0 = 0$, it is the environment.

Evolution
A membrane system evolves in the following way : It starts with the multisets of objects specified by the strings $w_1, \ldots, w_m$ in the corresponding regions. There is a global clock that ticks at every time step. At each step, in each

region, a multiset of objects and a multiset of rules is chosen and assignment of objects to rules is made. This is done in a *nondeterministic maximally parallel* manner. That is to say, no more rule can be added to the multiset of rules, because of the lack of objects and if there is a conflict of two rules for same (copy of an) object, then they are chosen non-deterministically. For example, if $w_i = a^5b^6c$ and the rules in R_i are $aab \rightarrow a_{here}b_{out}c_{in_2}$ and $aa \rightarrow a_{out}b_{here}$, then 2 copies of a can be assigned to one copy of first rule. Next two copies could be either assigned to another copy of first rule or to the second rule. And this choice is non-deterministic. Maximally parallel means that once four a's are consumed, no other rule can be applied to remaining objects. Thus non-deterministically, the result would be either a^3b^6c or a^2b^6c. It is important to remember that a^5b^6c represents the multiset $\{a, a, a, a, a, b, b, b, b, b, b, c\}$.

The output of a successful computation is the number of objects present in the output region in a configration such that no further evolution is possible. (i.e. no rule applicable in any membrane). Such a configuration is called halting configuration. A vector of the number of objects of each kind (e.g, no. of a's, no. of b's, etc) can also be taken as the output.

1.3 Illustrative Example

We here give the example of a membrane system to generate the Parikh image of the language $\{a^{2^n} \mid n \geq 0\}$. This system uses some ideas not defined above. Its actually called an evolution-communication P system. Consider the system :

$$\Pi = (O, \mu, w_1, w_2, R_1, R_2)$$

where
$\mu = [_1[_2]_2]_1$,
$O = \{a\}$,
$w_1 = \lambda$ and $w_2 = a$,
$R_1 = \{a \rightarrow a_{out}\}$ and $R_2 = \{a \rightarrow aa, (a \rightarrow aa)\delta\}$,
Here, δ at the end of a rule stands for "dissolution" of a membrane. This means, if a rule with δ at the end is used in a particular membrane, then that membrane will be dissolved after all the objects in that membrane evolve according to other (non-deterministically chosen) rules. The output of the system is the number of objects sent out to the environment.

As can be observed, we can either double the number of a's or dissolve the membrane 2 and stop; for, even if we apply $a \to aa$ on some a's and , $(a \to aa)\delta$ on some, the total number of a's is still doubled (objects evolve with the possible rules before the membrane is dissolved). So at some step, we get 2^n a's in membrane 1, from where they are sent to the environment (and hence contribute directly to the output). Only difference is that the computation halts there. So there is a non-deterministic choice of continuing or halting the computation. When we take the number of objects sent to environment as result, we naturally get the Parikh image of the language $\{a^{2^n} \mid n \geq 0\}$.

1.4 Variants of P systems

We here introduce two variants of P systems, which are of interest to us. We do not present here their formal definitions, but rather explain the intuition or central idea behind each of them, in order to avoid unnecessary jargon. These two models of our interest, which are P systems with worm objects and spiking neural P systems will be explained in detail in the following chapters.

1.4.1 P systems with worm objects

This model is inspired from the structure of DNA molecules (called as worms) present in the cell. They can be represented by a string. So our object of operation here, is a string, rather than a single symbol. A little formatting of the objects gives us this variant. Instead of operating on multisets of symbol objects, we operate on sets of string objects. This change in the format of objects makes it necessary to change the format of rules also. The kind of operations one deals with is a bit different here.

Basically we have four types of operations here, namely, replication, splitting, recombination and mutation. Strings can be split and recombined just like the splicing operation on DNA molecules. Replication and mutation are essentially the same operations as in DNA computing. For example, a simple rule like $a \to a_{here}a_{out}$ replicates the symbol object a. The only difference here is that this operation is done on string objects. Mutation rules are context free rules of type $a \to u$. Note that the rules are still defined with a being a single symbol. A string object containing this symbol can evolve using this rule. But if the string has more than one occurrence of a, only one *a* will be affected, and not all as in the basic model. Because here, its the string on which we operate, not the symbols. (The strings in basic model represent multisets of symbol objects). More about this in chapter 2.

1.4.2 Spiking Neural P systems

This model is inspired from the cell interconnection network and not the structure of a cell itself. In all other models, we consider the compartments of a single cell (separated by membranes) as “reactors”. But here, the compartments are characterized by cells themselves.

There is only one kind of object in spiking neural P systems – its called a spike. The system is represented by a graph whose vertices represent the cells (neurons) and edges represent the synapses that join two cells. Each neuron contains certain number of spikes initially. The evolution rules specify for each neuron, the number of spikes it is required to contain, in order to “fire’, and also the number of spikes it produces once it fires. When a neuron fires, it sends a single/multiple spikes to all the other neurons to which it has an edge. A computation stops when no neuron can spike any more (either all are empty or the number of spikes they have is not enough for them to spike). The result of a successful computation is measured either as the number of spikes in the output neuron, or as the number of spikes sent to the environment, or the distance (in time) between consecutive spikes. More about this in chapter 3.

1.5 Organization of the report

This thesis is organized as follows. In the next chapter we formally define P systems with worm objects and discuss their universality. We conclude this chapter with an important result regarding the universality (of P systems with worm objects), obtained by us. The next two chapters are dedicated to the investigation of the role of synchronization in the universality of SN P systems. In chapter 3, we define a variant of SN P systems which introduces a probabilistic asynchronism into the standard (synchronized) model, and discuss its reliability. We describe improvements in reliability obtained by experimentation, only to conclude that asynchronous systems may not be as powerfull as synchronous ones. Here, we come across the question - what is the compensation for this loss in power due to loss of synchronization?

We try to answer this question in the fourth chapter. We find out that asynchronous SN P systems can be made universal if the rules can emit up to four spikes at a time. We also come up with a hierarchy of asynchronous SN P systems with bounded number of neurons (1, 2, 3, etc.) and compare the power with corresponding system in synchronous mode.

Chapter 2

Universality of P systems with worm objects

One of the two main variants of P systems which are the topic of this project, is P systems with worm objects, introduced in [1] and further investigated in [2].

2.1 Introduction

Here we consider the type of P systems in which the objects are symbols as well as strings. The strings are also called as worms. The rules in this type of systems deal with strings (symbol objects are treated as strings of unit length). These type of P systems were introduced in [1]. The result of a computation is the number of strings in the output membrane. The strings are processed by four types of operation, viz. replication, splitting, recombination and mutation. Replication and splitting can increase the number of string objects whereas recombination and mutation cannot. These operations are discussed in detail in the following sections.

2.1.1 Formal Definition

P System with worms:
[Definition] A P system of degree $m \geq 1$ with worm-objects is a construct :

$$\Pi = (V, \mu, A_1, \ldots, A_m, (R_1, S_1, M_1, C_1), \ldots, (R_m, S_m, M_m, C_m), i_0)$$

where

- *V is an alphabet*
- *μ is a membrane structure consisting of m membranes arranged in an hierarchical structure*
- *A_i are finite multisets over V^*, associated with the corresponding regions of the structure μ.*
- *R_i, S_i, M_i, C_i are finite sets of replication, splitting, mutation rules and the set of objects used as crossing over blocks in the recombination operation. (These rules are explained in detail in the next section).*
- *$i_0 \epsilon \{1, 2, \ldots, m\}$ is the output membrane*

2.1.2 Operations

1. Replication $r : (a \rightarrow u_1||u_2; tar_1, tar_2)$ or $(a \rightarrow u_1||u_2; tar_1, tar_2)\delta$, where $a \,\epsilon\, V,\ u_1, u_2 \,\epsilon\, V^+$. This type of rule can be applied on a string w of the form x_1ax_2. It results in two strings, by replacing the occurrence of a in w by u_1 and u_2, respectively. We say that $w_1 \Longrightarrow_r (w_2, w_3)$, if $w_1 = x_1ax_2$, $w_2 = x_1u_1x_2$ and $w_3 = x_1u_2x_2$. After application of this rule, the strings w_2 and w_3 are sent to the membranes indicated by tar_1 and tar_2, respectively. The optional δ at the end of the rule specifies the action of dissolving the membrane. If a rule of that form is used, the membrane is dissolved at the end of that step. (We will discuss what this means in detail later).

2. Splitting $r : (a \rightarrow u_1 : u_2; tar_1, tar_2)$ or $(a \rightarrow u_1 : u_2; tar_1, tar_2)\delta$ where $a \,\epsilon\, V,\ u_1, u_2 \,\epsilon\, V^+$. Again this type of rule can be applied on a string w of the form x_1ax_2. It results in two strings – obtained by cutting w at a, and replacing the a by u_1 in one part and by u_2 in the other part. That is, we say $w_1 \Longrightarrow_r (w_2, w_3)$, if $w_1 = x_1ax_2$, $w_2 = x_1u_1$ and $w_3 = u_2x_2$. Also, the strings w_2 and w_3 obtained in this way are sent to membranes specified by tar_1 and tar_2, respectively, as before. Again the optional δ at the end specifies membrane dissolution action.

3. Mutation $r : (a \rightarrow u; tar)$ or $(a \rightarrow u)\delta$. This is a context free rule. That is, for a string containing a, this rule can be applied to obtain a different string by replacing a by u. We say $w_1 \Longrightarrow_r w_2$, if $w_1 = x_1ax_2$, $w_2 = x_1ux_2$. The string w_2 thus obtained, is sent to the membrane specified by tar, as before. And δ is the membrane dissolution action.

4. Recombination $r : (z; tar_1, tar_2)$ or $(z; tar_1, tar_2)\delta$. This type of rule can be applied to a pair of strings (say w_1 and w_2) which have z as their common substring. The operation involves "splicing" the strings w_1 and w_2 at z and recombining their splices in a crossed manner. That is we say $(w_1, w_2) \Longrightarrow_r (w_3,\ w_4)$ if $w_1 = x_1zx_2,\ w_2 = y_1zy_2$ and $w_3 = x_1zy_2,\ w_4 = y_1zx_2$ As before, the strings w_3 and w_4 are sent to membranes specified by tar_1 and tar_2, respectively. And δ means the membrane is dissolved after the current step.

Note that in all four type of rules above, the strings $x_1,\ x_2,\ y_1,\ y_2\ \epsilon V^*$. Note also that mutation rules can delete symbols because the string $u\ \epsilon\ V^*$. Also, replication and splitting rules can increase the total numbers of worms/strings, but recombination and mutation cannot.

We will now see what the membrane dissolution operation means and how does a P system with worms evolve.

2.1.3 Principles for evolution

The $(m+1)$-tuple $(\mu, A_1, \ldots, A_m)$ constitutes the initial configuration of the system. Starting with this, the system can pass from one configuration to another according the following principles :

- The work of the system is synchronized. That is, in each step, in each region, all strings that can be processed by a rule in that region are processed. This is called maximal parallel approach. Also, if more than one rules are applicable to a particular string (or a rule is applicable to more than one string) then the choice is done non-deterministically. So, the strings in all membranes are assigned to rules in their respective membranes in a non-deterministic, maximally parallel manner. And after the assignment is complete, the system evolves one step.

- A particular object (i.e. a particular copy of a string) can be processed by only one rule at any time, although different copies of the same string can be assigned to different rules, if possible.

- The strings resulting from the various operations are sent to the membranes specified by the targets in that rule. *here* means the string is not moved anywhere. *out* means that the resulting string is to be sent to the region surrounding the one in which this rule is applied. Similarly, in_j means the string is sent inside, to the membrane numbered j, provided, that membrane is adjacent to the present membrane. That is, we can only send strings between adjacent regions.

- When a rule containing the symbol δ is applied, the current membrane is dissolved. This means that the objects in this membrane are left free in the membrane surrounding it, and the rules of this membrane will be lost. The skin membrane can never be dissolved. In the event of a membrane dissolution, first all objects evolve according to the rules assigned to them (and may be are sent to other membranes also) and then the membrane is dissolved. For example, if during a step, a membrane i has objects $\{(a,1),(bb,2),(c,1)\}$ and rules $(a \to f\ ; out)$ and $(b; here, in_j)\delta$ then a evolves to f, is sent out of membrane i; the two copies of bb are recombined using the second rule, one is kept here and the other sent to membrane j, and then membrane i is dissolved.

2.2 Examples

2.2.1 Example 1 : Generating the length set of the language $\{a^{n!}\}$

Here we give a nonuniform solution. That is, the number of membranes in this P system will be dependent on the factorial number that you want to generate. Suppose we want to compute $n!$. Consider the following P system with $n+2$ membranes.

$$\Pi = (V, \mu, A_1, \ldots, A_{n+1}, (R_1, S_1, M_1, C_1), \ldots, (R_{n+1}, S_{n+1}, M_{n+1}, C_{n+1}), n+2)$$

Where
$V = \{b_i | 0 \leq i \leq n\} \cup \{x_i | 1 \leq i \leq n\}$,
$\mu = [_{n+1}[_n \cdots [_1]_1 \cdots]_n [_{n+2}]_{n+2}]_{n+1}$,
$A_1 = \{(b_0, 1)\}$, $A_i = \phi$, $2 \leq i \leq n+2$,

For all i such that, $1 \leq i \leq n$, the sets of rules for membrane i are as

follows :
$R_i = \{(b_{i-1} \to b_{i-1}||b_i; here, out)\}$ and
$M_i = \{(x_1 \to x_2; here), (x_2 \to x_3; here) \ldots (x_{n-i} \to x_{n-i+1}; here),$
$(x_{n-i+1} \to \lambda)\delta\}$

Finally, the rules for membrane $n+1$ are :
$M_{n+1} = \{(b_n \to b_n; in_{n+2})\}$
There are no other rules in the system.
Evolution. How does this P system work? It has initially 1 b_0 in membrane 1. From membrane i, it sends out as many b_i's as $n-i+1$ times the number of b_{i-1}'s that it has. This is because, in membrane i we count from 1 to $n-i+1$ by using the objects x_i and till then, we keep sending as many b_i's out, as the number of b_{i-1}'s we have. Hence, n b_1's are sent out of membrane 1. Then that is multiplied by $n-1$ (put $i=2$ in $n-i+1$) in membrane 2. This quantity is multiplied by $n-3$ in membrane 3 (similarly) and so on.

Finally, in membrane $n+1$ we get number of b_n's equal to $n!$, which are then sent to the output membrane by the rule $(b_n \to b_n; in_{n+2})$.

2.2.2 Example 2 : Generating the Fibonacci numbers

Consider the P system Π defined as follows :

$$\Pi = (V, \mu, A_1, A_2, A_3, (R_1, S_1, M_1, C_1), (R_2, S_2, M_2, C_2), 1)$$

Where,
$V = \{f_1, f_2, f, x, y\}$,
$\mu = [_1[_2]_2[_3]_3]_1$,
$A_1 = \phi$, $A_2 = \{(f_1, 1)\}$, $A_3 = \{(f_2, 1), (x, 1)\}$
$R_1 = \{(f_2 \to f||f_1; here, in_1)\}$, $M_1 = \{(f_1 \to f; here), (f \to f_2; in_2), x \to y; in_1\}$,

$R_2 = \phi$, $M_2 = \{(f_1 \to f_1; out), (y \to \lambda; out)\delta\}$,

$R_3 = \phi$, $M_3 = \{(f_2 \to f_2; out), (f_2 \to f_2; out)\delta\}$,

$S_2 = S_3 = C_2 = C_3 = \phi$.
Evolution. Lets examine the evolution of this system. Its based on the simple recurrence to calculate Fibonacci numbers : $f_n = f_{n-1} + f_{n-2}$. Membrane 2 stores objects of type f_1 and membrane 3 stores objects of type f_2

(initially one each). Both of them send their objects as such to membrane 1 i.e. out. Membrane 1 is our output membrane. If the computation halts at a step, it has number of f's equal to some n^{th} Fibonacci number.

Initially membrane 2 and membrane 3 have one f_1 and one f_2 respectively. They send them to membrane 1, where the f_1's are mutated to f's straight away, whereas the f_2's are replicated as f's here (membrane 1) and as f_1's in membrane 2. All the f's in membrane 1 are now sent back to membrane 2, as f_2's. And the addition can continue. This is nothing but following the recurrence. In short if $n(x)$ denotes the current number of objects of type x in membrane 1, and $n'(x)$ denotes the new number (i.e. after this step), then $n(f) = n(f_1) + n(f_2)$, and then we put the new values as $n'(f_1) = n(f_2), n'(f_2) = n(f)$, so that we always add the most recent two numbers in the sequence.

At any time, we can decide to stop this computation by applying the rule $(f_2 \rightarrow f_2; out)\delta$, which dissolves membrane 3. When this happens, the object x waiting there from the first step is set free in membrane 1, where in the next step itself the rule $(x \rightarrow y; in_2)$ is applied and the object y is produced in membrane 2. This y now causes membrane 2 to be dissolved by the rule $(y \rightarrow \lambda; out)\delta$ and the new number of f_1's – $n'(f_1)$ which is equal to $n(f_2)$ also adds to the current number of f's to give the next Fibonacci number in sequence and the computation stops.

Observe that there is a choice to dissolve membrane 3 but dissolution of membrane 3 automatically triggers the dissolution of membrane 2 and stops the computation after calculating one more Fibonacci number in sequence.

2.3 Known universality results

One of the best results known thus far in this area, due to Paun, et.al. [2] is as follows : $NCP_m = NRE$, for all $m \geq 6$.
This has been proved by constructing a worm-objects P system with 6 membranes that simulates a type-0 grammar in Kuroda Normal Form. The main challenge here is to sense the context. That is, for a context sensitive rule $AB \rightarrow CD$, we must check if the A in the input is indeed followed by a B, to be able to apply this rule. This has been achieved with help of replication and splitting operations.

Earlier in [1], universality was obtained with n membranes to simulate an n-matrix grammar in binary normal form. This was done with a straightforward construction. In what follows, we present a proof that four membranes suffice for universality. We have not proved that P systems with less than 4 membranes have strictly less power than Turing Machines but this certainly is one step toward the goal of finding/proving the optimality of this number.

2.4 Obtained Result

As is pointed out in [2], we try to come up with an optimal size P system (with worm objects) which is universal. Following is a proof that up to 4 membranes suffice for universality.

We consider a type-0 grammar in Kuroda normal form and construct a P-system using worm objects with four membranes to simulate this grammar.

As is known, the only context-sensitive rules in a grammar in Kuroda normal form are of the type $AB \to CD$. The context-free rules can be simulated straight away by the mutation rules in P systems. Our only concern is the context-sensitive rules.

To sense the context, we cut the string (representing the current sentential form) at the non-terminal A and send the two pieces inside a membrane, from where they can come out only if that A was actually followed by B in the original sentential form. This is the correct simulation of the context-sensitive rules. We need two membranes for doing this, one outer membrane for all context-free rules and one output membrane. So we need four membranes in all. Lets now define our system. Consider a type-0 grammar $G = (N, T, S, P)$ in Kuroda normal form. Suppose the context-sensitive rules in P are labelled $1, \ldots, r$. Consider the P system as defined below:
[Definition]

$$\Pi = (V, \mu, A_1, \ldots, A_4, (R_1, S_1, M_1, C_1), \ldots, (R_4, S_4, M_4, C_4), i_0)$$

where

- $\mu = [_1[_2[_3]_3]_2[_4]_4]_1$,
- $V = N \cup T \cup \{\Gamma, \dagger\} \cup \{B_i | AB \to CD \textit{ is labelled } i\}$,
- $A_1 = \{(S, 1)\}$, where, S is the initial symbol of G,
  $A_3 = \{(CDB_i, 1) | AB \to CD \ \epsilon \ P \textit{ is labelled } i\}$,

- $R_1 = \{(a \to \dagger||a; here, in_4)|a \epsilon T\}$,
- $S_1 = \{(A \to CDB_iB_i : DB_i; in_2, in_2)|AB \to CD \;\epsilon\; P \; is \; labelled \; i\}$,
- $M_1 = \{(A \to x; here)|A \to x \; is \; a \; context \; free \; rule \; in \; P\} \cup$ $\{B_i \to \lambda|1 \leq i \leq r\}$,
- $M_2 = \{(B \to B_i; here)|AB \to CD \;\epsilon\; P \; has \; label \; i\}$,
  $C_2 = \{(DB_iB_i; out, in_3)|AB \to CD \;\epsilon\; P \; is \; labelled \; i\}$
- $C_3 = \{(CDB_j; here, here)|AB \to CD \;\epsilon\; P \; has \; label \; j\}$,
- $M_4 = \{(A \to \Gamma; here)|A \;\epsilon\; N \cup \{B_i|1 \leq i \leq r\}\} \cup \{\Gamma \to \Gamma; here\}$

Evolution
As can be seen, the context-free rules are simulated straight away by mutation rules in membrane 1. For the context sensitive rules, we cut the current sentential string at A, to simulate the rule numbered j, $AB \to CD$. But we indicate the rule number in both the fragmented parts. That is, a string like w_1Aw_2 is cut into $w_1CDB_iw_2$ and DB_iB_i.

These parts are sent to membrane 2. Where all B's which can be at the second place in the left side of a context-sensitive rule can evolve to the number of their production. Note that if some non-terminal occurs in more than one rules, multiple such mutations are possible. But that has no effect. Because only if the B which was just next to the A in question was replaced by B_j, can the two fragments recombine and come out. Note that even if somewhere down in the string, by coincidence, D happens to be followed by B, replacing that B by B_j will not affect our simulation. This is because we need DB_jB_j as the recombining block, which is never possible anywhere else in the string. The coincidence just discussed produces only DB_j. So in all these cases, the fragments just can not come out of membrane 2 and the computation has no result.

And if in the current sentential form, A was indeed followed by B, then we recombine the two fragments using the rule $(DB_jB_j; out, in_3)$ in membrane 2, corresponding to this rule. That is if the original sentential form was $w_1ABw_2^{'}$ ($w_2 = Bw_2^{'}$, then the fragments that came in here were $w_1CDB_jB_j$ and $DB_jBw_2^{'}$ and they recombine to produce $w_1CDB_jB_jw_2^{'}$ and DB_jB_j. We intend to send the latter string inside membrane 3, which is used as a storage. Nothing happens to these strings in membrane 3. But if the recombination is done the other way, DB_jb_j could be sent out and the other string could come in membrane 3. Should that happen, the computation never halts, because, the string coming in has CDB_j as its substring so it can be recombined forever with the string CDB_j waiting here from the beginning. Hence only in the case expected, we can continue. B_j is mutated to λ in membrane 1 and we get $w_1CDw_2^{'}$ in membrane 1 finally, which is the correct simulation of the rule $AB \to CD$.

Once we get a string of terminals, we use replication rules in membrane 1 and sent as many objects to the output membrane (i.e. membrane 4) as there are terminals in the string. If these rules are used in between, then a string containing a non-terminal (or B_i) comes in and it produces the trap symbol Γ which does not allow the computation to halt.

To conclude, our approach is exactly similar to what Gheorghe Păun has done in [2]. We have just come up with a more polished result that reduces the number of membranes.

2.5 Future work

Obviously, finding out if a P system with 3 membranes can characterize RE languages is the main topic of further investigation. However, P systems with less than 3 membranes are very less likely to have the power of sensing context. So, we expect a negative answer here.

But a question beyond this is, can we avoid replication rules altogether? In general replication is a very powerful operation, and it would be interesting to find a universal system without using replication. This needs further investigation.

Chapter 3

Reliability of Spiking Neural P systems

We intuitively defined spiking neural P systems in section 1.4.2 of chapter 1, as the model of P systems inspired by the functioning of neural cells. Here, we define formally the Spiking Neural P systems and explain the roll of synchronization in their evolution. Two variants of SN P systems are defined, depending upon whether synchronization is present or not. We then move on to define an extension to this basic model, the Stochastic spiking neural P systems, obtained by introducing a probabilistic asynchronism. We define formally the stochastic spiking neural P systems (SSN P systems for short, [3]) and define their reliability. The SSN P systems allow us to introduce a controlled asynchronism in the system and observe, by way of experimentation (for example, simulation) the effect of increasing asynchronism on the reliability of the system.

We describe the methodologies we used (and the results we got) to get more reliable SSN P systems, in a quest to come up with powerfull (or in the best case, universal) asynchronous systems.

3.1 Introduction

3.1.1 Formal definition of an SN P system

[Definition]
An SN P system is a quadruple

$$\Pi = (O, \Sigma, syn, i_o)$$

Where:

1. $O = \{a\}$ is the singleton alphabet (a is called a spike)
2. $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ are neurons of the form

   $$\sigma_i = (n_i, R_i),\ 1 \leq i \leq m$$

   where :

   - $n_i \geq 0$ is the initial number of spikes contained by the neuron
   - R_i is a finite set of rules of the following two forms :
     (a) $E/a^r \rightarrow a; d$; E is a regular expression over O, $r \geq 1$ and $d \geq 0$;
     (b) $a^s \rightarrow \lambda$ for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule of type (a) in R_i;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ is a set of synapses among the neurons
4. $i_o \in \Sigma$ is the output neuron

Evolution
Clearly, the system is represented by a graph whose vertices are the neurons and edges are the synapses between them. A global clock is assumed to tick at each step and in each step, all neurons that have at least one enabled rule, have to fire. Hence the model is synchronous.

The objects (there is only one type of object:a) evolve by means of spiking rules of type (a) as above. If the number of spikes contained in a neuron is described by the regular expression E (i.e. the neuron contains c spikes, say, and $a^c \in L(E)$), then the rule $E/a^r \rightarrow a; d$ is enabled in that neuron. Once the rule fires, it consumes r spikes (i.e. $c - r$ spikes remain in the neuron) and sends out one spike to all the neurons it is connected to, after d time steps. That is, if the rule fires in time step t, then in steps $t, t+1, \ldots t+d-1$, the neuron is closed, so that it cannot receive new spikes. All spikes sent to this neuron during these steps are lost. In step $t + d$, the neuron spikes, and becomes open again. Note that, once a neuron fires, the spikes sent out to its neighbors reach them immediately and the updation of number of spikes in all open neurons (including itself) happens instantaneously.

A rule of type (b) above is called a forgetting rule. And it is enabled only when the total number of spikes contained in a neuron is exactly equal to s. When this rule fires, all the s spikes contained in this neuron are consumed. Note that the delay plays no significant role here, as only the contents of this neuron are affected. Hence we omit the delay in these types of rules.

We observe that the working of an SN P system is sequential at a neuron's level. That is, in each neuron, at each step, if there are more than one rules enabled by its current contents, then only one of them (chosen non-deterministically) can fire. But still, the system as a whole evolves parallely and synchronously, as in, at each step, all the neurons (that have an enabled rule) choose a rule and all of them fire at once.

Starting from the initial configuration $C_0 =< n_1/0, \ldots n_m/0 >$, where n_i/t_i means that the neuron σ_i contains n_i spikes, and will open after t_i spikes, the system evolves as described above. As suggested in [4], we can associate a number (or a vector of numbers) with a computation in several ways. One of the most frequently associated numbers is the distance in time between two spikes sent to the environment by the output neuron. Note that, only those computations where the output neuron spikes exactly twice are successful.

3.1.2 Asynchronous SN P systems

The asynchronous SN P systems differ from the synchronous ones defined above in a subtle manner. If we remove the assumption of synchronization from the above definition, we get asynchronous SN P systems. In an asynchronous SN P system, there is no global clock. Each neuron acts on its own discretion. So, in a given configuration, if a neuron has some rules enabled, it can choose to fire or remain silent. It is not obligatory for an enabled neuron to spike. Moreover, during this time (for which a rule is enabled, but not fired) the neuron is open. It can receive new spikes. And if new spikes are deposited, the computation continues in the new scenario. The same rules may no longer be enabled now. Again, the neuron can choose to fire one of the newly enabled rules, or remain silent.

Note that the delay parameter d has no meaning in asynchronous systems, because there is no global clock. So the spiking rules in asynchronous SN P systems do not have this component. Also, the output of a computation is the "total" number of spikes sent to the environment by the output neuron. Hence, the output neuron can spike any number of times, unlike in the synchronous model.

3.1.3 An Illustrative Example

Consider the SN P system $\Pi = (O, \Sigma, syn, i_o)$ as shown below :

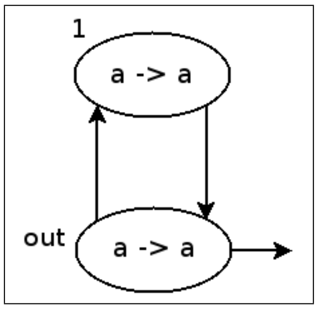

Figure 3.1: An SN P system where synchronization matters

Where :
$\Sigma = \{\sigma_1, \sigma_{out}\}$, $syn = \{(1, out)\ , (out, 1)\}$, $i_o = \sigma_{out}$ and $C_0 =< 1, 1 >$.
Synchronous Evolution: Under the assumption of synchronization, Π works forever. Both neurons use a rule in each step. So, output neuron sends one spike out in each step. We get an infinite spike-train, written 1^{ω}, as the output.
Asynchronous Evolution: Without the synchronization assumption, the system can halt at any moment. Each neuron can wait an arbitrary number of steps before using its rule. If both neurons fire simultaneously, the computation continues, otherwise, one neuron consumes its spikes and the other gets two, so can never spike. Hence, the system generates the set N.

From the above example, we observe that it is not very clear which of the systems - synchronous or asynchronous - are more powerful. With two neurons, synchronous systems can only generate the set FIN, but asynchronous systems can generate infinite languages also. We continue this comparison in chapter 4.

3.2 Stochastic Spiking neural P systems

We saw that the synchronization is a rather important assumption while defining SN P systems. Cavaliere et. al. [3] defined an extension of the SN P systems, called SSN P systems, which is obtained by introducing a probabilistic asynchronism into the basic SN P systems. Thus, these systems lie in between the synchronous and asynchronous systems. We define these systems here and then discuss the relevance with our problem.

3.2.1 Formal definition

An SSN P system is same as a standard SN P system, with the difference that, with each spiking rule, a probabilistic delay is associated, instead of a deterministic delay d as defined in last section. The probabilistic delay is described by a probability distribution function $F^{'}(.)$ (for a spiking rule), or $F^{''}(.)$ (for a forgetting rule). Once a rule is enabled in a neuron, a random amount of time decided by the probability distribution function $F^{'}(.)$ or $F^{''}(.)$ associated to that rule elapses before that rule spikes. If two rules get enabled and fire simultaneously, conflict is resolved non-deterministically, as before. A computation is said to halt when no rule is enabled in any neuron. We can again associate a number with the halting computation in several ways, as described in previous section. The set of numbers generated in such a way by all halting configurations is the language generated by the SSN P system.

Observe that in SSN P systems, after a rule is enabled, a time interval (whose value is determined by a probability distribution) ellapses before it spikes. For this time interval, a neuron can still receive new spikes from other neurons. Due to these new spikes, the former rule may no longer be enabled. The computation continues in the new scenario. This is unlike synchronous SN P systems (and like asynchronous SN P systems).

In a sense, the probability distribution chosen for a rule determines what time it will take to fire, once enabled. Clearly, the degree of asynchronism depends upon the distribution. This relation of the probability distribution with the asynchronous behavior of the system is our topic of interest. Cavaliere et.al. [3] suggested an approach to evaluate the effect of increasing asynchronism on the system's ability to perform correctly (i.e. simulate synchronous behavior) by way of simulation using the Mobius modeling framework ([5]). We take this approach further to answer some of the questions posed there. This experimentation and results are explained in section 3.3. Before proceeding to see the results, lets understand the probabilistic asynchronism by way of an example.

3.2.2 Example

We modify an example (of an SN P system) from [4] to form an SSN P system by using ideas from [3] as shown in figure 3.2 below:

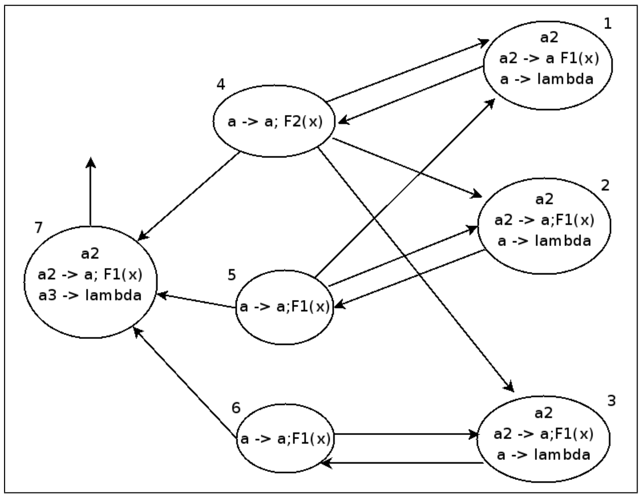

Figure 3.2: Probabilistic asynchronism : An SSN P system

Where :

- $F_1(x)$ is the Gaussian normal distribution $N(\mu_1, \sigma^2)$ with $\mu_1 = \sigma^2 = 0$
- $F_2() = 0.5H(x) + 0.5H(x-1)$, i.e. the discrete uniform distribution in $\{0,1\}$., where $H(x) = 1$ if $x = 0$, 0 otherwise.

First lets understand the probability distribution functions $F_1(x)$ and $F_2(x)$. The first function is a Gaussian normal distribution with mean and variance equal to zero. No variance means the probability of mean is 1. Hence, x

takes value zero with probability one. The second function returns 0.5 for $x = 0$ and 0.5 for $x = 1$, zero for all other values of x. Hence, both the values 0 and 1 are taken with equal probability.

Initially, only neurons 1, 2, 3 and 7 have spikes, two each. So, they fire in the first step. So one spike is sent to the environment by neuron 7 during the first step only. And in 1,2 and 3, the first rule $a^2 \rightarrow a$ is enabled and fires immediately with probability one (thats how F_1 is defined). They send one spike to neuron 4,5,6 respectively. In neurons 5 and 6 the only rule (which returns the spike received) is enabled and also fired immediately with probability. But in neuron 4, the only rule which is enabled, fires after either zero time or one time unit. Each of these two happens with same probability - 0.5. This is just a simulation of non-deterministic choice. If the rule fires immediately, each of neurons 1, 2 and 3 receive 2 spikes while neuron 7 receives 3 spikes (one from each 4, 5, 6). Neurons 1, 2 and 3 continue their execution as before and neuron 7 forgets the three spikes received by the second rule in it.

In case neuron 4 decides to fire the rule after 1 time unit, neurons 5 and 6 send their spike to neurons 1,2 and 3, where they are forgotten by the second type of rule. Whereas, neuron 7 receives 2 neurons (one from each of 5 and 6) which it consumes to send out one spike to environment. After one time unit, the prepared spike from neuron 4 comes and is forgotten in neurons 1,2, and 3, while remains unattended in neuron 7. Thus the computation halts. Suppose neuron 4 fired the only rule immediately for n times before deciding on delaying it. Clearly, at time $2n + 2$, computation halts.

Here, we take as output the distance (in time) between first two spikes sent to the environment. Remember that the first spike was sent at time 1. Hence the distance between first two spikes is $2n + 2$. Also note that the computation always halts with the second spike. Hence we generate the numbers of the form $2m$.

3.2.3 Reliability

As can be observed, the choice of probability distribution function greatly helps us in simulating a controlled and synchronous behavior. Had we chosen a non-zero variance in our example from section 3.2.2, we would not get the firing time of zero with probability one. And that would certainly have affected the system behavior. That is, we loose synchronization. But how much does it affect the language generated?

Cavaliere et. al. ([3]) proved that SSN P systems become universal if we can choose the firing time distributions. This was shown by simulating register machines. If in this proof, we change the variance to some non-zero value, will the resulting system still be able to simulate the register machine correctly always? If not always, then what is the probability of correct simulation?

Correct simulation of an instruction

A register machine has only three types of instructions, namely $ADD(r)$ (deterministic and non-deterministic), $SUB(r)$, and $HALT$. An ADD instruction is supposed to add one to the contents of register r. So, an SSN P system Π simulates correctly an ADD instruction if started in the same configuration as the register machine, Π ends up in a configuration corresponding to this incremented value of register r, keeping the contents of all other neurons unaffected (except for the ones representing next instruction).

Correct simulation of SUB and $HALT$ can be defined similarly. In [3], it was shown that as the variance of firing time distributions increases, the probability of correct simulation of an instruction decreases. In sections 3.3.2 and 3.3.2, we investigate how this probability can be improved.

Formal definition of reliability

Reliability of an SSN P system Π simulating a register machine M, $R_{\Pi}^{M}(n)$ is defined as the probability that Π simulates correctly a sequence of n instructions executed by M, when M starts from the initial configuration and Π starts from the corresponding one [3].

Our aim here is to define/re-define SSN P modules that simulate a given register machine more reliably, under increasing variance (i.e. increasing asynchronism).

3.3 Improvements in reliability of SSN P systems

3.3.1 Questions

The obvious question we ask is, how do we come up with models that have a high probability of correct simulation of an instruction with increasing variance (and hence increasing asynchronism). But there is more to it. Once we rewrite the modules corresponding to ADD, SUB etc to improve this probability, how does it affect the reliability of the system as a whole? Basically, we want to address the following questions:

- Can the topology of the network influence the reliability, or can suggestions from real biological networks of neurons be taken to improve reliability?
- Can redundancy (number of neurons/connections per module) improve reliability? How much redundancy? How to make use of it?
- Is there a possibility of associating exponential distributions to firing times instead of normal, so that the models can be solved analytically to investigate reliability?
- In general, when is it possible to implement powerful computing devices with high degree of asynchrony? That is, what is the trade-off for loss of synchronization?

3.3.2 Methodologies and results obtained

We here address the questions asked in section 3.3.1. We have not been able to come up with answers for last but one question i.e., possibility of associating exponential distributions and solving the models analytically. We describe here some ways to improve reliability by way of the first two suggestions instead. For each of our suggestions, we re-define the corresponding SSN P system and simulate using the Mobius modeling tool [5] to calculate the reliability.

Suggestions from biology

Cavaliere and Mura asked in [3] if topology of the network could affect the reliability. That is, without concern to the specific language the system is generating or the specific purpose of the system, can certain topologies be found to be more reliable than others? In [6], it was shown that this indeed is true in case of biological networks. Biological networks are highly reliable systems composed of asynchronous, unreliable elements. This reliability is achieved through the abundance of certain subgraphs and suppression of others. So reliability in these systems strongly depends on the underlying topology. The three 3-node subgraphs shown in figure 3.3, which allow for reliable dynamics, were observed to be abundant in nature.

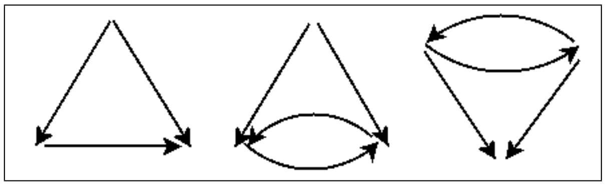
Figure 3.3: Topologies that allow reliable dynamics

But these topologies could not be incorporated into our models of ADD and SUB instructions preserving all the other assumptions and without changing the model semantics at large. Defining entirely new models, targeted at making use of these "reliable" subgraphs as components is one important future work.

Redundancy

In [3] itself it was conjectured that redundancy could help by showing that adding one neuron to the ADD module indeed improves the reliability. We go ahead and take this suggestion for SUB module as well to find out that this is true for up to four neurons. That is, if we go on adding neurons in the success branch of SUB module, the reliability goes on increasing, up to four neurons, there onwards, it remains constant. We link this to one conclusion drawn in [6] – that "any type of feed-forward wiring yields reliable dynamics and from any initial condition the system reaches a fixed point after a short while". The difference in reliability obtained by adding up to three neurons in this way appears in the graph shown in figure 3.4.

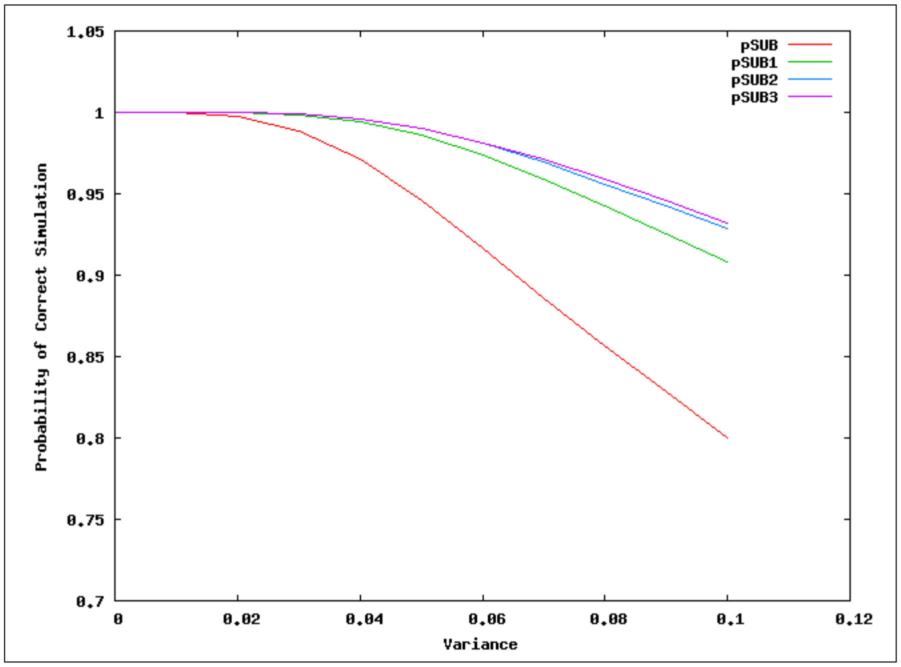

Figure 3.4: Reliability of SUB module for increasing redundancy

As can be observed, only the first improvement is significant. All others can not be counted. So it seems that redundancy helps, but the best way to use redundancy is to use a feed-forward connection with one extra neuron. Interestingly, for the ADD module, even the second extra neuron doesnot yield any improvement. The plot corresponding to this is not included here to avoid repitition.

Reducing Unreliability

We observed that the reliability calculated largely depends on the method used by Mobius to arrive at it. What the simulator does is that in each state, it calculates the set of possible next states and all enabled rules. It then samples the firing time distributions of these rules and updates the state of the network according to the spiking times. This generates different possible trajectories in the state space. Now the reward is calculated for all the trajectories and then the mean is returned. So if we define an impulse reward on an activity and that activity never fires in a trajectory, then the reward collected for this trajectory will be zero.

We next observe that in the ADD module the problem occurs if the two neurons adding two spikes in the register neuron fire at different times, then both of them are consumed one by one and addition does not happen. We define a new version of ADD module by adding one more neuron (as already shown in [3] and adding one more rule to it - $a \to \lambda$. What it does is, if at all the two spikes are consumed one by one in the register, this neuron also does the same thing and blocks the computation. So this computation does not produce any output. So there are less chances of the system producing wrong output.

Now, this improvement can not be captured by the notion of reliability because of reasons explained earlier. For this reason, we define the reward in the reverse way. It now returns one when the activity fires, and still addition has not happened. Thus, in a way, we are calculating the unreliability of the system. What is the probability that it proceeds with wrong computation. Figure 3.5 shows the improved unreliability of ADD module with our modification.

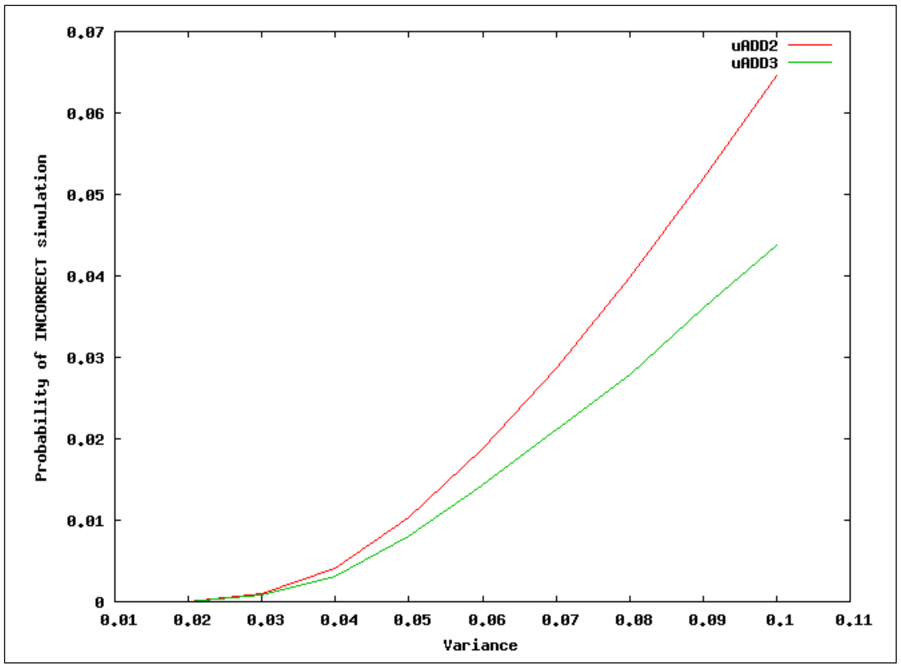

Figure 3.5: Probability of incorrect simulation of ADD

We also note that the reliability of this new model as calculated with mobius appears to be slightly lesser than the model in [3]. But this is because in our model, there still can be cases where the two spikes are consumed one by one in the new neuron whereas addition has happened in the register neuron (the forgetting rule there did not spike until the second spike came). In such cases, earlier model does not block computation whereas our model does. But as discussed before, these computations anyway do not produce any output. So they are in a sense harmless.

3.4 Conclusion

In general, it seems that coming up with very reliable and at the same time, asynchronous SN P systems is a bit difficult. The experimentation hints at the conclusion that asynchronous SN P systems are probably not as powerful as the synchronous ones. Then comes the last question from section 3.3.1. What is it that makes up for this loss in power? We try to find the answer in the next chapter.

Chapter 4

Universality and decidability of asynchronous SN P systems

As concluded in the last chapter, from the experimentation, it looks like asynchronous SN P systems are probably not as powerful as synchronous ones. But, as already pointed out, asynchronous SN P systems (like the synchronous ones) also become universal, when using extended rules ([7]). Extended rules are spiking rules that can emit more than one spike. That is, spiking rules of the form $E/a^r \rightarrow a^p$, such that $r \geq p$.

So, if we prove that asynchronous SN P systems using only standard rules are not universal, we can say that the programming ability given by extended rules makes up for the loss in power due to loss of synchronization. Hence, we will probably answer the last question in section 3.3.1. On the other hand, if it turns out that asynchronous SN P systems using standard rules are also universal, then the whole problem is closed. But it is hard to prove it either way. We here prove that asynchronous SN P systems using extended rules of a limited form (i.e. they can emit only up to four spikes) can become universal, and we also give an intuitive argument that with less than length four, it is unlikely to happen.

The difference in power between synchronous and asynchronous systems is still not clear from the above result. We further try to compare the two modes by comparing corresponding systems with a limited number of neurons. To this end, we define a hierarchy of asynchronous SN P systems with increasing number of neurons and compare the results with synchronous systems.

4.1 Known results

Various models of SN P systems defined till now have been proved to be universal (or non-universal). We enlist the results known thus far. We denote by $Spik_2P_m^{syn}(rule_k, cons_p, forg_q)$ the family of sets of numbers computed by synchronous SN P systems with at most m neurons, using at most k rules in each neuron, consuming at most p spikes in any spiking rule and forgetting at most q spikes in any forgetting rule.

We denote by $Spik_{tot}P_m^{nsyn}(\gamma, del_0)$, $\gamma \in \{gen, unb, boun\}$, the family of sets of numbers computed by asynchronous SN P systems using at most m neurons (replaced by $*$ if there is no bound on this) of type γ (general, unbounded or bounded) and where the output is defined as "all" (*tot* means total) spikes sent out by the output neuron.

- $Spik_2P_2^{syn}(rule_*, cons_*, forg_*) = NFIN$. [4]
- $Spik_2P_*^{syn}(rule_2, cons_3, forg_3) = NRE$. [4]
- $Spik_2P_*^{syn}(rule_3, cons_3, forg_3, bound_3) = SLIN$ (Bounded neurons with up to 3 spikes). [4]
- $Spik_{tot}EP_*^{nsyn}(gen, del_0) = NRE$. [7]
- $N_\mu Spik_{tot}EP_*^{nsyn}(unb, del_0) = NPBCM$. [7]

It is clear that when using only unbounded neurons (and not general) in asynchronous mode, we get a sub-universal class. The case of interest is when using general neurons in asynchronous systems.

4.2 Obtained result

We want to prove that asynchronous SN P systems using extended rules of length up to four (i.e. those that can emit up to four spikes) are universal. Asynchronous SN P with extended rules of arbitrary length have already been shown to be universal. So this proof will demonstrate that extended rules that can emit up to four spikes are sufficient. We also informally argue that it is not possible to prove this using extended rules of length less than four. It once again underlines the conjecture that asynchronous SN P systems with standard rules are not universal, because standard rules are nothing but extended rules with length 1. Here, we actually prove that we can simulate matrix grammars with appearance checking (a Turing-equivalent class of devices) using asynchronous SN P systems with extended rules of length up to four.

4.2.1 Extended rules of length up to four give universality

To prove that asynchronous SN P systems can simulate matrix grammars with appearance checking using λ-rules.

Consider a matrix grammar with appearance checking in binary normal form, $G = (N, T, S, M, F)$ such that $N = N_1 \cup N_2 \cup \{S, \#\}$ with these three sets mutually disjoint. We know that the matrices in M can be in one of the following forms :

1. $(S \to XA),\ X \in N_1,\ A \in N_2$
2. $(X \to Y,\ A \to x),\ X, Y \in N_1,\ A \in N_2$ and $x \in (N_2 \cup T)^*, |x| \leq 2$
3. $(X \to Y,\ A \to \#),\ X, Y \in N_1,\ A \in N_2$
4. $(X \to \lambda,\ A \to x),\ X, Y \in N_1,\ A \in N_2$, and $x \in T^*$

We construct an asynchronous SN P system Π to simulate the above grammar G. The SN P system Π uses extended rules of length up to four. The two main challenges involved are : 1) To ensure that either both of the rules of a matrix are applied or none are and 2) To correctly simulate the rules in appearance checking mode. We attack both of these by using extended rules of length up to four.

Construction

Let us formally define the asynchronous SN P system constructed as above.

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$$

with m neurons. Each non-terminal X is represented by one coordinating neuron and k other neurons, where k is the number of rules in which the non-terminal X occurs. These neurons are named $\sigma_X, \sigma_{X_1}, \sigma_{X_2} \ldots \sigma_{X_k}$, respectively. So, each σ_{X_i} represents the enabler for a rule in which X appears.

Also, there are sixteen auxiliary neurons per matrix $m \in M$. Apart from all these, there is a neuron σ_{out}, designated as the output neuron. At any time, it contains number of spikes equal to the length of the terminal string generated by grammar G.

The output of the system is defined as the number of spikes contained in σ_{out} when the system is in a halting configuration (that is, when all neurons are open but none is firable).

The initial state of the system is described by $n_X = 2, n_A = 2$ and $n_Z = 0$ for all other non-terminals $Z \in N_1 \cup N_2$. That means, the coordinating neurons of non-terminals X and A contain two spikes each, and all other neurons contain no spikes, where, $(S \rightarrow XA)$ is the first matrix.

Now, the coordinating neuron for each non-terminal plays an important role. Whenever a particular non-terminal (say Z) is present in the current sentential form, the coordinating neuron corresponding to that non-terminal (i.e. σ_Z) will have 2 (or multiples of 2, depending upon how many Z's are present) spikes. Neuron σ_Z consumes two of these spikes (representing the rewriting of one Z) and passes them non-deterministically to one of the subordinate neurons, each of which represents a rule in which that particular non-terminal occurs. This simulates the non-deterministic choice among the rules applicable to the current sentential form. However, it does not check whether the particular matrices in which these Z-rules appear are applicable in full or not.

Simulating non-appearance-checking matrices

For matrices of type 2 (non-appearance checking), as discussed above , the main challenge is ensuring that either both the rules or none are executed. For this, we use a common neuron to which both the neurons representing these rules (as discussed above) spike. Consider, for example, a matrix $m = (X \rightarrow Y,\ A \rightarrow BC)$ of type 2. So here, the neurons σ_{X_i} and σ_{A_j} will spike to the common neuron belonging to the matrix m, where $X \rightarrow Y$ is the i-th X-rule and $A \rightarrow BC$ is the j-th A-rule. This common neuron simulates the action of both the rules, if and only if both of the neurons spike. If just one of them spikes, then the computation never halts, and if none of them spike, then the spikes flowing through this part of the system are just forgotten in this common neuron. For example, the actions of the rules in m are simulated by adding two spikes to coordinating neurons of Y, B, C, that is the neurons σ_Y, σ_B and σ_C.

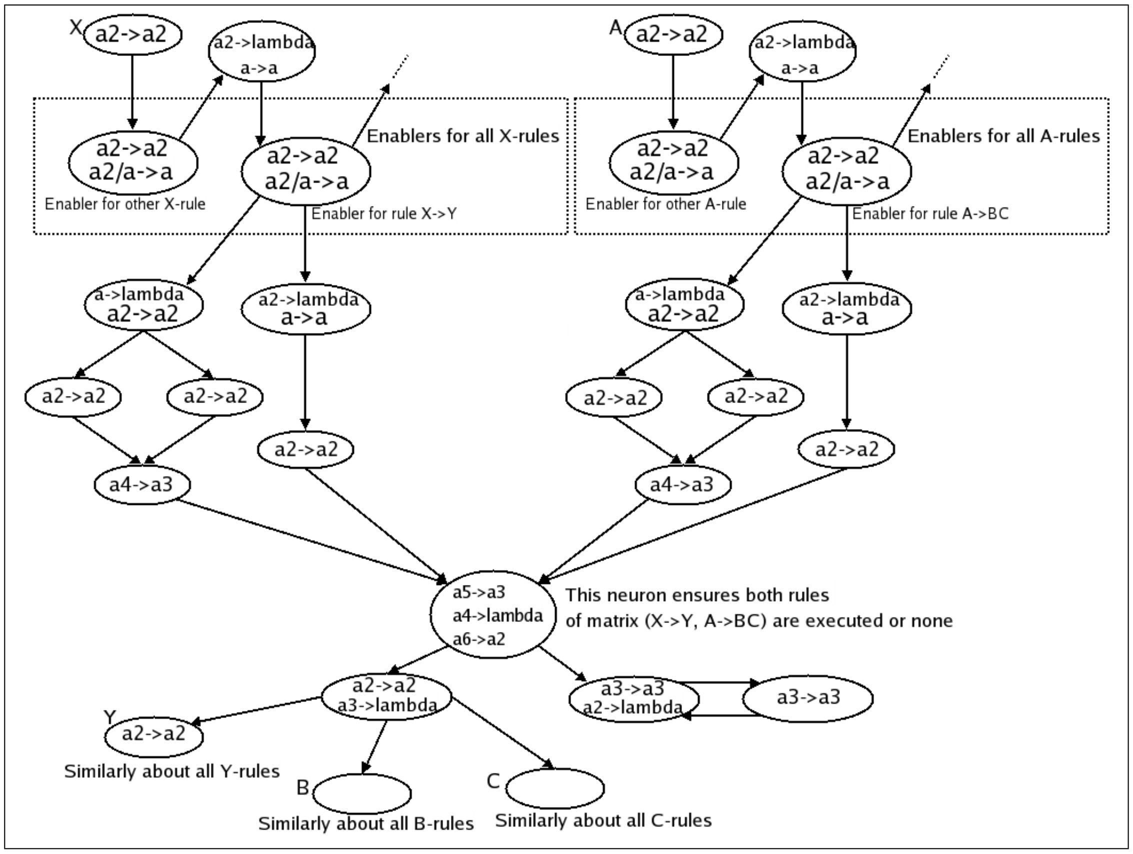

Figure 4.1: Asynchronous SN P module to simulate non-appearance-checking

Note however that we define the "spiking" and "not spiking" of these neurons (σ_{X_i} and σ_{A_j}) a bit differently. Namely, a neuron has "spiked" if it sends out 3 spikes and has "not-spiked" if it sends out 2 spikes. This is needed because we can't detect the absence of a signal in the asynchronous systems due to the unpredictable time delay before the signal actually arrives. We actually argue informally in section 4.2.2 that it is this inability of asynchronous systems that makes use of extended rules unavoidable.

The module (that is, part of the asynchronous SN P system) that simulates a matrix of type 2 is shown in figure 4.1.

Simulating appearance-checking matrices

For applying matrices of type 3 (appearance checking), we have to ensure the "absence" of a certain non-terminal in the current sentential form. For example, a matrix $(X \rightarrow Y, A \rightarrow \#)$, if applied when A is present, traps the computation and if applied when A is absent, just converts X to Y. In other words, if the coordinating neuron corresponding to A - σ_A - has (a multiple of) 2 spikes, we have to scrap the computation, that is, make it loop forever. And if does not contain (multiple of) 2 spikes, we have to let the execution continue. So we make a structural change here. We do not use a sub-ordinate neuron under the neuron σ_A to represent the rule $A \rightarrow \#$. Instead, we take a spike directly from σ_A. Further, we change the rules in the common neuron so that, it allows the computation to proceed only if the neuron σ_{X_i} (representing the first rule in the matrix being simulated) spikes. If both the neurons σ_{X_i} and σ_A spike, then we make the computation loop forever as before. If the neuron σ_{X_i} does not spike, we do not care about the presence or absence of A and we just forget the spikes flowing through the system, and let the computation proceed.

Remember again, that "spiking" and "not spiking" of a neuron is defined a bit differently, as before. Also note that we could directly take a connection from σ_A, without conflicting with other A-rules, because we know that the appearance checking and non-appearance checking rules on a certain non-terminal do not conflict (either one is used or the other).

Figure 4.2 describes a module (part of the asynchronous SN P system) that simulates a matrix of type 3.

Simulating terminal matrices

For matrices of type 4, the same module as in figure 1 can be used, with the only difference that, for each terminal a on the right side of the rule $A \rightarrow x$, we deposit a spike in the output neuron, to account for the length of the terminal string generated.

Claim - The grammar G is simulated correctly by Π.

Proof:

From the construction, it is clear that a computation is possible in Π iff it is possible under rules of G. Also, the output neuron contains one spike per terminal generated during this computation. Hence, a number n can be generated using Π, iff a string x with $|x| = n$ can be generated using G.

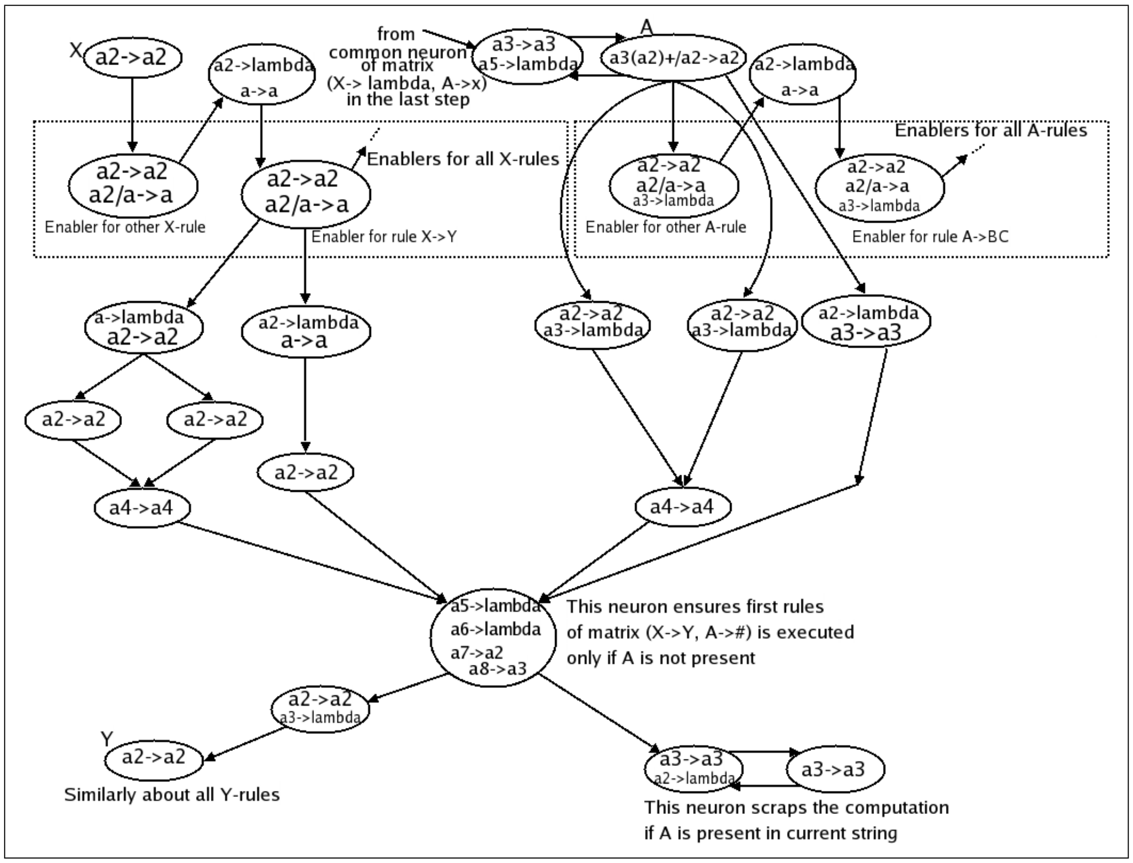

Figure 4.2: Asynchronous SN P module to simulate appearance-checking

Hence, the proof.

4.2.2 Is it possible with less than four?

We have proved that asynchronous SN P systems using extended rules capable of emitting up to four spikes are universal. We also observe that in asynchronous systems, we can not distinguish the absence of a signal (spike) from the delay in its arrival. So, we have to define each (presence and absence) with a different number of spikes. Hence, use of extended rules is unavoidable. Now, with extended rules of length only 2, we can assign 2 spikes to mean presence and 1 spike to mean absence. But in this case, when we want to detect presence of signals coming from two independent sources, absence of both signals, and presence of only one signal are both represented by 2 spikes and hence indistinguishable. Adding an independent source doesnot help, since the same argument applies to the signal coming from there. Hence, one of the two sources has to be able to emit more than 2 spikes, in which case, we can assign 3 spikes to mean presence and 2 to mean absence.

But even in this case, appearance checking (detecting the presence of one particular signal and absence of the other) is not possible, by a similar argument, because we can not differentiate between the scenarios (signal1 present, signal2 absent) and (signal1 absent, signal2 present) with the above assignment. Hence, we have to be able to emit 4 spikes from a single source so that we can assign 4 to mean signal1 present, 2 to mean signal1 absent, 4 to mean signal2 present and 3 to mean signal2 absent. One can see that all four cases are distinguishable with this assignment.

4.3 A hierarchy

We saw that asynchronous SN P systems using extended rules capable of emitting up to four spikes become universal. But apart from the intuitive argument in the last section, we could not produce a formal proof that the systems with standard rules (or extended rules capable of emitting less than four spikes) are not universal. We instead try to define a hierarchy by restricting the number of neurons, starting with 1 neuron, 2 neurons and so on. To do this, we borrow ideas from [8], where, a similar argument, although in a different context (asynchronous systems used as language generators) was recently made. We realize that the parameters like number of neurons, length of extended rules, number of rules per neurons are intricately related and one can compensate for the power of other in its absence and vice versa. This was already pointed out (and proved for some of the parameters) in [9]. Following are the results.

4.3.1 One neuron can generate at most FIN

For synchronous SN P systems, it was argued in [4], that with 1 neuron, any finite set can be generated. Synchronous systems make use of delays for this. The only neuron spikes at the first step and then, non-deterministically spikes after n_i steps, for all n_i in the finite set to be generated. Here, we do not have delays. For asynchronous systems also, one thing is clear, that with one neuron, we can not generate more than a finite set of numbers.

To prove that any finite set can be generated, we have to take help of extended rules. If we can have extended rules that can emit any number of spikes, we can generate any finite set F with a single neuron, by including the following rules in the only neuron available :

1. $a^{n_{max}} \rightarrow a^{n_{max}}$
2. $a^{n_{max}}/a^{n_i} \rightarrow a^{n_i}$; $a^{n_{max}-n_i} \rightarrow \lambda$, $\forall n_i \in F$

Where, $F = \{n_1, n_2, \ldots, n_{max}\}$ is the finite set to be generated and n_{max} is the maximum number in the set F. As can be seen, the neuron can non-deterministically choose either the first rule (in which case it generates n_{max}, or one of the first rules of type 2 to generate any n_i and consequently forget the remaining spikes.
Thus we conclude :

$$Spik_{tot}P_1^{nsyn}(gen, del_0) \subseteq FIN.$$

$$Spik_{tot}EP_1^{nsyn}(gen, del_0) = FIN.$$

4.3.2 Two neurons can generate at most REG

The synchronous SN P systems can generate only finite sets, even with two neurons. This is because, the output neuron can spike at most twice in a successful computation. So, the number of spikes in the system can not increase and any computation can last for a finite number of steps only.

In asynchronous SN P systems, however, the output neuron can spike any number of times, so there is a possibility of generating infinite sets. But, with two neurons, replication of spikes is not possible. So the computation can continue only by means of exchanging spikes in between the two neurons, which means, the number of possible different configurations is finite. The configurations serve as the state of an automata. So, passing through a finite set of states, the system can generate at most regular languages. We extend it to our model and context. It is not clear if two neurons have the power to generate any given regular language in the asynchronous mode. We get the following result.

$$Spik_{tot}P_2^{nsyn}(gen, del_0) \subseteq REG.$$

4.3.3 Three neurons can generate at least REG

We want to prove that any regular language L can be generated by three neurons working in the asynchronous mode. Consider a regular grammar $G = (N, T, S, P)$, where $N = \{A_i | 1 \leq i \leq n\}$, $S = A_n$ and P contains rules of the form $A_i \to bA_j$, for some $b \in T$ and $L = L(G)$. We construct the asynchronous SN P system Π with 3 neurons as shown in figure 4.3 to simulate G.

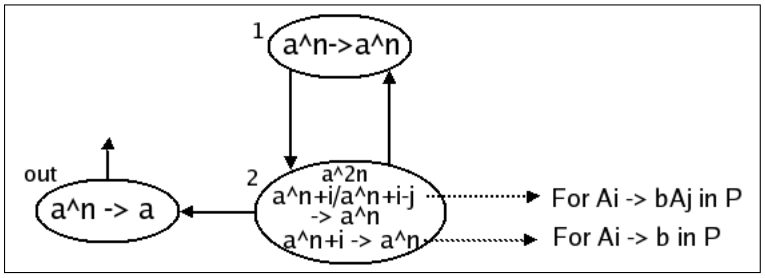

Figure 4.3: Asynchronous SN P of 3 neurons to simulate regular grammars

It can be seen that Π indeed simulates G correctly. The neuron σ_{out} sends out one spike per terminal symbol generated. The neuron σ_2 actually simulates the rules and neuron σ_1 is an auxiliary neuron used to temporarily store the n spikes. Note that we make use of extended rules of arbitrary length here. We conclude that :

$$REG \subseteq Spik_{tot}P_3^{nsyn}(gen, del_0).$$

4.3.4 Four neurons can generate a non-semilinear language

Consider the following asynchronous SN P system with 4 neurons.

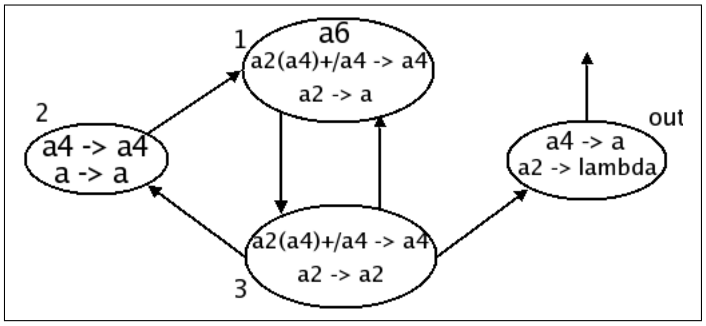

Figure 4.4: Asynchronous SN P of 4 neurons to generate $\{a^{2^n}\}$

Neuron σ_1 initially spikes and sends 4 spikes to σ_2 as well as σ_3. Thus replication is used to double the number of 4-spike chunks in the system. σ_3 collects these doubled 4-spike chunks and when the remaining two spikes come, it fires and sends out all these chunks to σ_{out} and back to σ_1. σ_{out} now sends one spike to the environment per chunk of 4 spikes received from σ_3. Finally, when σ_3 sends out last 2 spikes, the output neuron forgets them, and the next iteration is triggered in σ_1. The doubling can continue any number of times. But as the system is asynchronous, it can happen that σ_2 does not spike until σ_1 spikes for the second time and then it acquires 5 spikes. In this situation, σ_1 can never spike again and the computation stops. The generated number is of the form $2^{n+1} - 2$, if the computation continues for n steps. That this set is non-semilinear is obvious.

A summary appears in table 4.1.

No. of neurons →	**1**	**2**	**3**	**4**	*****
Synchronous	FIN	FIN	?	?	RE
Asynchronous	$FIN^{\dagger}$	$\subseteq REG$	$\supseteq REG^{\dagger}$	$\supset REG^{\dagger\dagger}$	$RE^{\dagger\dagger}$

Table 4.1: A Hierarchy : Synchronous v/s Asynchronous SN P

Remarks :

† : Extended rules (of unbounded capacity) needed.

†† : Extended rules capable of emitting up to 4 spikes needed

4.4 A Decision Problem for Asynchronous SN P systems

We first construct a conditional grammar with context-sensitive rules which simulates the work of an asynchronous SN P system. Although it does not generate the exact language of the SN P system, it generates this language modulo a projection. It turns out that this grammar is useful in arguing that membership problem for asynchronous SN P is semidecidable.

4.4.1 Simulation using Conditional grammars

To prove that conditional grammars with context sensitive rules can simulate asynchronous SN P systems, consider an asynchronous SN P system

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$$

with m neurons and one output neuron σ_{out}. Each neuron σ_i contains n_i number of spikes and R_i is the set of standard spiking rules in it. Let's label each spiking rule as follows : Rule number j in neuron σ_i is labeled r_j^i (Assume, within a neuron, we assign a particular numbering to the rules).

We construct a conditional grammar $G = (N, T, S, P)$ to simulate Π as follows :

$$N = \{S\} \cup \{A_i | 1 \leq i \leq m\}$$

$$T = \{a, B\}$$

and the set of context-sensitive (length-increasing) productions is specified as follows : $S \to A_1^{n_1} A_2^{n_2} \ldots A_m^{n_m} \in P$,
For each rule r_i^j (i.e. rule number j in neuron σ_i), we add the following productions according to the form of the rule :

- **Case 1** : $r_i^j : a^x/a^y \to a$ (A special case is when x=y). Then, we add the rule (r, R) where, r is $A_i^y \to B^y A_{j_1} A_{j_2} \ldots A_{j_n}$, such that, $(i, j_k) \in syn, 1 \leq k \leq n$ and $R = A_1^* A_2^* \ldots A_i^x \ldots A_m^* B^*$.

- **Case 2** : $r_i^j : a^x(a^y)^*/a^z \to a$. Then, we add the rule (r, R) where, r is $A_i^z \to B^z A_{j_1} A_{j_2} \ldots A_{j_n}$, such that, $(i, j_k) \in syn, 1 \leq k \leq n$ and $R = A_1^* A_2^* \ldots A_i^x (A_i^y)^* \ldots A_m^* B^*$.

- **Case 3** : $r_i^j : a^x \to \lambda$. Then, we add the rule (r, R) where, r is $A_i^x \to B^x$, and $R = A_1^* A_2^* \ldots A_i^x \ldots A_m^* B^*$.

Apart from these, we add to P, the following rules :

- (r_1, R_1) such that r_1 is $A_iA_j \rightarrow A_jA_i, 1 \leq i < j \leq m$ and $R_1 = (N)^* - A_1^*A_2^* \dots A_i^* \dots A_m^*B^*$.
- (r_2, R_2) such that $r_2 = BA_i \rightarrow A_iB$ and $R_2 = (N \cup \{B\})^*$.
- $A_{out} \rightarrow a$; $A_i \rightarrow B$ for $1 \leq i \leq m$, all with a common permitting regular expression: $V^* - \bigcup_{i=1}^{k} R_i$, where R_i's are permitting regular expressions of all the other (say k) as above.

This last set of rules is enabled, when none of the above rules are enabled; this indicates the halting condition of the SN P system in question.
Claim - The grammar G simulates Π.
Proof - The initial state of the system is represented by the first sentential form (derivable using the only rule with S on the left hand side), $A_1^{n_1}A_2^{n_2} \dots A_m^{n_m}$. So each non-terminal A_i represents the number of spikes contained in the corresponding neuron σ_i. Now assume that a neuron σ_i has synapses to σ_j and σ_k.

For a bounded rule $a^x/a^y \rightarrow a$, we need to enable this rule when there are x spikes in neuron σ_i. That is to say, we need to be able to delete y A_i's and add one A_j and one A_k, when the current sentential form has x number of A_i's. That is exactly what is done by rules added in case 1 above. The only difference is, we do not have deletion here. So these rules produce a blank symbol to signify deletion.

For an unbounded rule like $a^x(a^y)^*/a^z \rightarrow a$, we can extend the same approach, because we can use the regular expression $a^x(a^y)^*$ directly in the enabling part of the corresponding grammar rule.

For forgetting rules, we don't need to add a spike to σ_j and σ_k. We only need to produce x blank symbols if the rule is $a^x \rightarrow \lambda$.

There is one problem, though. When we use any spiking rule, the A_j and A_k symbols produced are not in their right place. In fact they are near the A_i's. We need to rearrange them. Also, the blank symbols produced to signify deletion have to be moved to the end of the string. We use the last two types of productions to do this. The first type is used to rearrange A_i's. But this need only be done till all the A_i's come together (for all i). This is indicated by the enabling regular expression of this production. Also, the second production takes B's to the right end of the string. Once all B's are moved to the right end, the rule can not be applied anyway. So its enabling regular expression is not needed.

So after applying a spiking rule each time, one has to use these rules a number of times to once again bring the sentential form into the form $A_1^* A_2^* \ldots A_i^* \ldots A_m^* B^*$. The execution continues just like the SN P system except for these intermediate rearrangement steps.

Parallelism and Asynchronous behavior - What do you mean by parallelism? In an SN P system, if σ_i and σ_j both produce spikes into σ_k, then parallel execution means that before σ_k can execute/enable any rule, both σ_i and σ_j should have been done with their execution. But this is true only in case of synchronous behavior. In the asynchronous case, one of σ_i and σ_j can choose to remain silent, and hence σ_k can continue without waiting for both of them. So to simulate this behavior with a sequential grammar, all we need to do is we need to give both of these choices – execution sequence where σ_i and σ_j are both done before σ_k starts, and execution sequence where this is not true. And this is indeed given by the non-deterministic choice of enabled productions. In a given sentential form, all the productions whose corresponding spiking rules would have been enabled in the corresponding state of SN P system, are enabled by virtue of the permitting regular expressions. So, we believe that this grammar indeed captures the asynchronous parallel behavior of SN P system.

4.4.2 Membership Problem : Semidecidable

The membership problem for asynchronous SN P systems is stated as: Given an asynchronous SN P system Π and a number n, can Π generate the number n? We claim that this problem is semidecidable. To prove this, we use the context-sensitive conditional grammar constructed above. Although it does not directly generate the language of the simulated SN P system, it generates the language L, such that a string x of the form $B^* a^n B^*$ is in L if and only if the number n is $L(\Pi)$ – the actual language of the SN P system. We use this fact to produce the following algorithm to answer the membership question for asynchronous SN P systems.

Here we add a neuron σ_O to the given SN P system Π. This neuron collects the spikes sent out to the environment by the output neuron of Π. We construct a conditional grammar G to simulate this new asynchronous SN P system Π, as described in section 4.4.1. Note that the number of spikes in this newly added neuron σ_O can only increase. We are interested in finding if in any halting configuration of Π, the neuron σ_O contains n spikes. We propose to use the same algorithm as used to decide the membership problem for context-sensitive grammars, as defined in [10].

[Algorithm]

Compute the sequence of sets $V_i,\ i \geq 0$, iteratively defined by :

$$V_0 = \{S\},$$
$$V_{i+1} = V_i \cup \{\beta | \exists \alpha \in V_i, \alpha \Rightarrow_G \beta \text{ } and \text{ } |\beta|_{A_O} \leq n\}.$$

Where, $|\beta|_{A_O}$ denotes the number of A_O's in the string β and A_O is the non-terminal which represents the newly added neuron σ_O.

Note that in an asynchronous SN P system, for any given configuration, there are a finite number of neurons which are enabled (fireable) in that configuration. Lets say this number is m. Each of these m neurons can either spike or remain silent. Each of these 2^m spiking combinations can produce a different configuration. So the maximum number of configurations reachable from a given configuration is finite. These configurations are represented by the sentential forms of the grammar G. Hence, from a given sentential form α of G, we can reach a finite number of configurations β. This implies that the sets V_i are finite.

Also note that for each $i \geq 0$, for any $\alpha \in V_i,\ \beta \in V_{i+1}$, $|\beta|_{A_O} \geq |\alpha|_{A_O}$. That is, the number of A_O's can only increase. But what we do not know is whether this number *will* increase at each step. So the process of defining V_i's may never end. But, if it does end, we can answer the question.

Assuming the process stops at V_k, $n \in L(\Pi)$ if and only if, any of the strings $w \in V_k$, such that $|w|_{A_O} = n$ represents a halting computation of Π. In other words, $n \in L(\Pi)$ if and only if the last rules $A_i \rightarrow a_i$ (for all neurons σ_i) are applicable to any $w \in V_k$, such that $|w|_{A_O} = n$. This can easily be tested. As the condition is both necessary and sufficient, the problem is decidable, if the process ends. We conclude that the membership problem for asynchronous SN P systems is semi-decidable.

4.5 Conclusion

We have proved that extended rules capable of emitting up to four spikes produce universality, and argued informally that it is less likely to happen with less than four. We have also argued that membership problem for asynchronous systems is semidecidable. This indicates we have moved at least one step ahead toward proving that asynchronous SN P using standard rules are not universal.

Chapter 5

Summary and future work

We have investigated two variants of P systems – P systems with worm objects and spiking neural P systems. Below we summarize the results obtained and future work in both the cases.

- We have proved that P systems with worm objects with 4 membranes are universal. The problem of finding optimal value of number of membranes required for universality remains open for future work. Also one needs to investigate if replication rules can be avoided altogether.
- Although we have proved an upper bound on the capability of extended rules required to get universality, we do not yet know what is the lower bound. Thus, one possible direction for future work in this regard is to write an asynchronous SN P system (using standard rules) to simulate a universal class of computing devices, which at the moment seems to be difficult. On the other hand it would be interesting to prove that a subuniversal class can simulate asynchronous SN P systems. The work in section 4.4.1 might be useful in this regard.

Bibliography

[1] Juan Castellanos, Gheorghe Păun, and Alfonso Rodrn. Computing with membranes: P systems with worm-objects. In *String Processing and Information Retrieval*, pages 65–74, 2000.

[2] Gheorghe Păun and Carlos Martin-Vide. Computing with membranes: One more collapsing hierarchy. *Bulletin of the EATCS*, 72, 2000.

[3] Matteo Cavaliere and Ivan Murra. Experiments on the reliability of stochastic spiking neural p systems. *Technical Report 26/2007, The Microsoft Research-University of Trento Centre for Computational and Systems Biology*, July 2007.

[4] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural p systems. *Fundam. Inf.*, 71(2,3):279–308, 2006.

[5] D. Daly D. Deavours S. Derisavi J.M.Doyle W.H. Sanders P.Webster G. Clark, T. Courtney. The mobius modeling tool.

[6] Konstantin Klemm and Stefan Bornholdt. Topology of biological networks and reliability of information processing.

[7] Matteo Cavaliere, Mihai Ionescu, Gheorghe Păun, Oscar Ibarra, Omer Egecioglu, and Sara Woodworth. Asynchronous spiking neural p systems. *Proceedings of the 13th International Meeting on DNA Computing*, 2007.

[8] Xingyi Zhang, Xiangxiang Zeng, and Linqiang Pan. On languages generated by asynchronous spiking neural p systems. *Theor. Comput. Sci.*, 410(26):2478–2488, 2009.

[9] Oscar H. Ibarra, Andrei Păun, Gheorghe Păun, Alfonso Rodríguez-Patón, Petr Sosík, and Sara Woodworth. Normal forms for spiking neural p systems. *Theor. Comput. Sci.*, 372(2-3):196–217, 2007.

[10] Arto Salomaa and Alexandru Mateescu. *Aspects of Classical Language Theory*, volume 1 of *Handbook of formal languages*, pages 175–246. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[11] Arto Salomaa. *Formal Languages.* Academic Press, NY and London, University of Turku, Turku, Finland, first edition, 1973.

[12] Gheorghe Păun and Carlos Martin-Vide. Elements of formal language theory for membrane computing. *Reseach group on Matematical Linguistics, Report GRLMC 21/01.*

[13] S.N.Krishna. *Languages of P systems : Computabiliy and Complexity.* PhD thesis, IIT Madras, 2001.

[14] Jurgen Dassow, Gheorghe Paun, and Arto Salomaa. *Grammars with Controlled Derivations*, volume 2 of *Handbook of formal languages*, pages 101–150. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[15] Gheorghe Păun. Introduction to membrane computing. 1998.