MODELLING AND SIMULATION OF SN P SYSTEMS USING PETRI NETS

A THESIS

Submitted by

VENKATA PADMAVATI METTA

in partial fulfilment for the award of the degree

of

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THAPAR UNIVERSITY PATIALA

September 2012

Shuklambaradharam Vishnum Shashi Varnam Chaturbuhjam
Prasanna Vadanam Dhyaayet Sarva Vighnopashaantaye
Agajaanana Padmaarkam Gajaananam Aharnisham
Anekadantam Bhaktaanaam Ekadantam Upaasmahe

Sarasvati Namastubhyam Varade Kamarupini
Vidyarambham Karishyami Siddhirbhavatu Me Sada
Padmapatra Visalakshi Padmakesara Varnini
Nityam Padmaalayam Devi Sa Mam Pathu
Saraswathi Bhagavati Poornendu Bimbanana

DEDICATED

TO

MRS. K. VEERA LAKSHMI AND MR. K. SATYA NARAYANA
(PARENTS)

PROF (MRS). KAMALA KRITHIVASAN
(GUIDE)

THESIS CERTIFICATE

This is to certify that the thesis entitled **MODELLING AND SIMULATION OF SPIKING NEURAL P SYSTEMS USING PETRI NETS** submitted by **Venkata Padmavati Metta** to Thapar University, Patiala for the award of the degree of **Doctor of Philosophy** is a bonafide record of the research work carried out by her under our supervision and guidance. The content of the thesis, in full or parts have not been submitted to any other Institute or University for the award of any other degree or diploma.

Research Supervisors

Deepak Garg

Dr. Deepak Garg
Department of Computer Science and Engineering
Thapar University, Patiala

Kamala Krithivasan

Prof. Kamala Krithivasan
Department of Computer Science and Engineering
IITM, Chennai

Place: Patiala
Date:

Place: Chennai
Date:

Acknowledgements

During the course of my time as a Ph.D. student, I have been fortunate enough to benefit from the generosity and kindness of many people.

I express my deep gratitude to my research guide Prof. (Mrs.) Kamala Krithivasan for her untiring support right through my INAE mentorship program at IIT Madras, unto selection of topic to brick by brick assembly of this thesis. It is impossible to explicate the importance of her support over the past few years. I am deeply indebted to her for her patient and enthusiastic participation during my contact sessions and her valuable suggestions which made every seemingly complex problem a relative cakewalk. I cannot forget her care and affection I have received during my stay at IIT, Madras.

I am immensely thankful to my supervisor, Prof. Deepak Garg for his continuous guidance, suggestions and support without which I could not have accomplished this task. I am much grateful for his kind help and cooperation right from my admission to Thapar University unto my gradual progress semester by semester.

I am thankful to the management of Bhilai Institute of Technology, Durg especially to Shri. I.P. Mishra, Dr. M.K. Kowar and Dr. A. Arora for their extended support and providing me special casual leave for INAE mentorship program at IIT, Madras and for attending the research progress seminars at Thapar University.

I thankfully acknowledge the suggestions and encouragement I have received from the doctoral committee members of Thapar University.

I am also thankful to the Department of Computer Science and Engineering, IIT, Madras and Thapar University, Patiala for allowing me to utilize the research infrastructure during my stays at IIT, Madras and Patiala respectively.

I am thankful to Prof. Benedikt Löwe, Elsevier foundation, and Bhilai Institute of Technology, Durg for providing me financial assistance to attend the Computability in Europe-2010 conference at Ponta Delgada, Portugal.

My special thanks and indebtedness to Prof. Marian Gheorghe, University of Sheffield, UK from whom I received invaluable suggestions related to the topic of this thesis. Thanks also to PNetLab software team, for allowing me to utilize the software in this work.

The work has greatly improved because of the comments of the anonymous reviewers of the research papers which contributed to this dissertation. I record my thanks for them.

I would like to thank my family, especially my loving, supportive and encouraging husband for his support during this Ph.D.

I am much grateful to my colleagues at BIT who always smilingly extended their timely support. I offer my regards to all of those who supported me in all respects during the completion of the research.

Last but not the least, I thank the Almighty for reasons too numerous to mention.

PADMAVATI

Abstract

Natural computing deals with the extraction of mathematical models of computation from nature, investigating their theoretical properties, and identifying the extent of their real-world applications. *P systems* (also called *membrane systems*) were introduced as parallel computational models inspired by the hierarchical structure of membranes in living organisms and the biological processes which take place in and between cells.

Spiking neural P systems (for short, *SN P systems*) are a class of P systems inspired by the spiking activity of neurons in the brain. An SN P system is represented as a directed graph where nodes correspond to the neurons having spiking and forgetting rules. The rules involve the spikes present in the neuron in the form of occurrences of a symbol a. It is a versatile formal model of computation that can be used for designing efficient parallel algorithms for solving known computer science problems. SN P systems are used as a computing device in various ways - generators, acceptors, and transducers.

SN P system with anti-spikes (for short, *SN PA systems*) is a variant of SN P system containing two types of objects, spikes (denoted by a) and anti-spikes (denoted by $\overline{a}$), corresponding somewhat to inhibitory impulses from neurobiology. Because of the use of two types of objects, the system can encode the binary digits in a natural way and hence can represent the formal models more efficiently and naturally than the SN P systems.

The thesis investigates the computing power of spiking neural P system with anti-spikes as language generators and transducers. We show that SN PA systems as generators can generate languages that cannot be generated by the standard SN P systems. It is demonstrated that, as transducers, spiking neural P systems with anti-spikes can simulate any Boolean circuit and computing devices such as finite automata and finite transducers. We also investigate how the use of anti-spikes in spiking neural P systems affect the capability to solve the *satisfiability problem* in a non-deterministic way.

For efficient formalization and to deal with the implementation and formal correctness of SN P systems, this thesis also shows the structural link between SN P systems and *Petri nets*. A major strength of Petri nets is their support for analysis of many properties and problems associated with parallel systems. Petri nets working in sequential or parallel mode are also used as language generators.

The SN P system works in a locally sequential and globally maximal way. That is, each neuron, at each step, if more than one rule is enabled, then only one of them can fire. But still, all neurons fire in parallel at the system level. This makes it suitable for a natural translation to Petri net with parallel semantics. In general for arbitrary classes of P systems using maximal parallelism, translations to Petri nets are only possible through special semantics associated with them. In this thesis we are interested in those interactions investigating the role of Petri nets as a tool to express behavioural semantics for SN P systems.

The thesis proposes a direct translation of standard SN P systems, SN P systems with anti-spikes and extended SN P systems into Petri net models that exactly mimic the working of the systems on simulation. The Petri net models obtained after translation are considered for simulation using *PNetLab*. PNetLab is a Java based Petri net tool which supports the parallel execution of transitions. It also allows to write user defined guard functions in C/C++, which makes it possible to represent the regular expressions associated with spiking/forgetting rules. It also provides step-by-step system watching for collecting simulation reports.

We relate the languages generated by the SN P systems with the step languages generated by the corresponding Petri nets. We emphasize the relationship between spiking neural P systems and Petri nets by constructing SN P systems for *simplex stop-and-wait protocol* and *producer/consumer paradigm*. They are translated into equivalent Petri net models, which are observed as standard solutions based on Petri nets already present in the literature. It is attractive to adopt Petri nets to model SN P systems so the rich theoretical concepts and practical tools from well-developed Petri nets could be introduced in the current research of SN P systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis relates two important mathematical models: spiking neural P systems and Petri nets. In this chapter first we give a short informal survey of the assorted computing models of natural computing, especially spiking neural P systems. We also briefly mention the historical development of Petri nets and the relation between the P systems and Petri nets. We also discuss the motivation and the contribution of this thesis. Finally, we describe the organization of the thesis and the topics of each chapter.

1.1 Background

Natural computing uses nature as a source of inspiration or metaphor for the development of new techniques for solving complex computational and engineering problems. *Biologically inspired computing* and *computing with natural means* are the important branches of natural computing. Biologically inspired computing involves the study of biological phenomena, processes and even theoretical models for the development of computational systems and algorithms capable of solving

complex problems. Research in this field includes *artificial neural networks*, inspired by the functioning of the mammalian brain [64], *evolutionary algorithms* motivated by evolutionary biology [10], *swarm intelligence*, based on the collective behaviour of social organisms.

Computing with natural means is the approach that brings the most radical change in the paradigm. This field is motivated by the need to identify alternative media for computing. Researches are now trying to design new computers based on molecules, such as membranes, DNA and RNA, or quantum theory. The idea resulted in what is now known as *molecular computing* [4] and *quantum computing* [96] respectively.

Molecular computing is based upon the use of biological molecules to store information and genetic engineering techniques for the design of new computers. The field of DNA based computation [36, 86] laid the foundation towards molecular computing. The major idea of *DNA computing* is to take advantage of the huge parallelism provided by the biochemical processes occurring in a DNA solution.

Membrane Computing which is another branch of molecular computing initiated by Gh. Paun [28], provides distributed, parallel, and non-deterministic computing models known as *P systems*. These models are basically abstractions of the compartmentalized structure and parallel processing of biochemical information in biological cells. Membrane computing proved to be a fruitful framework for applications in several areas, especially in biology and bio-medicine [21]. Many P system variants have been defined in literature and many of them have been proven to be computationally complete. Moreover, several general classifications of P systems are considered depending on the level of abstraction: *cell-like* (a rooted tree where the skin or outermost cell membrane is the root, and its inner membranes are the children or leaf nodes in the tree), *tissue-like* (a graph connecting the cell membranes) [71], and *neural-like* (a directed graph, inspired by neurons interconnected by their axons and synapses).

Spiking neural P systems were introduced in [50] with the aim of defining computing models based on ideas specific to spiking neurons, currently much investigated in neural computing. The resulting models are a variant of *tissue-like* and neural-like P systems from membrane computing (see [30] and the up-to-date information at the web site [2]), with very specific ingredients and way of functioning.

In short, a standard SN P system consists of a set of neurons (cells, consisting of only one membrane) placed in the nodes of a graph and sending signals (spikes, denoted in what follows by the symbol a) along synapses (edges of the graph). Thus, the architecture is that of a tissue-like P system, with only one kind of object present in the cells.

One key reason of interest for P systems is that they are able to solve computationally hard problems (e.g. NP-complete problems) usually in polynomial to linear time only, but requiring exponential space as trade off. These solutions are inspired by the capability of cells to produce an exponential number of new membranes via methods like mitosis (membrane division) or autopoiesis (membrane creation). The website of the domain, at *http://ppage.psystems.eu/*, provides a comprehensive information in this respect.

It is usually a complex task to predict or to guess how a P system will behave. Moreover, as there do not exist, up to now, implementations in laboratories (neither *in vitro* nor *in vivo* nor in any electronic media), it seems natural to look for software tools that can be used as assistants that are able to simulate computations of P systems. The first software simulator for P systems appeared in the year 2000. It was written by Mihaela Malita [69] in LPA-Prolog and, since then, many software simulators have been presented (see [40] and references therein). Furthermore, software tools, such as P-Lingua [26], for simulating P systems, have been developed and used in real life problems. Their common purpose is the better understanding of the computational process of P systems, for pedagogical purposes as well as assistance for researchers.

The modelling and analysis of P systems has also attracted considerable interest from the Petri net community [55–60, 89]. A deeper investigation of the relationship between these two formalisms is interesting, providing valuable cross fertilization of these research areas. Membrane computing deals with the computational properties, making use of automata, formal languages, and complexity results. The formal model of *Petri nets* is a generalisation of automata theory such that the concept of concurrently occurring events can be expressed in a simple but powerful framework. The semantics of Petri nets is mathematically defined, and Petri nets have the advantage of being executable. Petri nets are also used as language generators [41].

Petri nets were first introduced in the early 1960s by Carl Adam Petri in his Ph.D. dissertation [85]. Since that time, Petri nets have been accepted as a powerful formal specification tool for a variety of systems including concurrent, distributed, asynchronous, parallel, deterministic, and non-deterministic systems. An article by Murata [76] contains a good introduction to general Petri net theory. Additionally, there exist a number of introductory articles and books on Petri nets [92]. For general access to information on Petri nets, see the Petri nets home-page *http://www.informatik.uni-hamburg.de/TGI/PetriNets/*.

It is well-known that the computational power of Petri nets is strictly weaker than that of *Turing machines*, making them inadequate for modelling certain real-world systems such as prioritized systems [5]. To overcome this shortcoming, a number of extended Petri nets have been introduced to enhance the expressive capabilities of Petri nets. Among them are coloured Petri nets [54], Petri nets with inhibitor arcs, timed Petri nets [90, 95], prioritized Petri nets, and more. Extended Petri nets are powerful enough to simulate Turing machines.

In our thesis we propose to use Petri nets as a tool to simulate, verify, and analyse different variants of SN P systems. Petri nets offer significant advantages because of their twofold representation: graphical and mathematical. They can be as

well a model of parallelism, where the simultaneity of the events is more important, when we consider their step sequence semantics in which an execution is represented by a sequence of steps each of them being the simultaneous occurrences of transitions [62, 87].

1.2 Motivation

It is obvious that the chemical, electrical, and informational processes taking place in the brain are the major source of inspiration for informatics. Risking a forecast, we believe that if something great is to appear in informatics in the near future, then it will be inspired by the brain. Spiking neural P systems are not the answer to this learning-from-brain challenge, but only to call (once again) the attention to this challenge. Becoming familiar with brain functioning, in whatever reductionistic framework (as spiking neural P systems investigation is), can however be useful.

Membrane computing is now an area of intense research related to applications, mainly in biology/medicine, but also in economics, distributed evolutionary computing, computer graphics, etc. [21], but this happens after a couple of years of research of a classic language-automata-complexity type; maybe this will be the case also for the spiking neural P systems, which need further theoretical investigation before passing to applications.

As SN P systems are recently introduced computational models, there are only a few software tools available to analyse their behaviour. In [39], a tool for simulating simple and extended SN P system is introduced that yields only the transition diagram showing the reachable configurations for the SN P system and it lacks graphical step-by-step simulation of the system.

Petri nets provide a promising way to verify system properties, system soundness and to simulate the dynamic behaviour of the SN P systems because of their parallel execution semantics, appropriateness to represent typical working processes and the availability of Petri net tools to simulate these systems. This was the driving force to motivate our research in this field.

While relating the languages associated with different variants of SN P systems and the corresponding Petri nets, the computational efficiency of different variants of SN P systems are explored. The power of different variants of SN P systems as language generators are investigated in [18, 19]. It was shown in [18] that some finite languages cannot be generated using simple SN P systems but it was proved in [19] that SN P systems with extended rules can generate the finite languages.

Standard spiking neural P systems are used to simulate arithmetic and logic operations where the presence of spike is encoded as 1 and absence of spike as 0 [73]. The negative integers however, are not considered. In transducer mode, they are used to simulate the Boolean circuits [52], with two spikes sent out of the system encoded as 1 and one spike as 0. In [66], a uniform solution to the SAT (in CNF, with n variables and m clauses) is provided using standard SN P systems without delay having $3n^2 + 8m + 5$ neurons, providing the solution in a number of steps which is linear in the number of variables. Two bits were used to code each literal of a clause, hence the computation cannot end in less than $2n$ steps.

The SN P systems with anti-spikes can encode 1 by a spike and 0 by an anti-spike in more natural and efficient way than standard SN P systems, which motivated us to study the computability of SN PA systems as language generators and transducers.

1.3 Contribution of the Thesis

In this thesis we design algorithms to translate standard SN P systems, SN P systems with anti-spikes and extended SN P systems into Petri net models. We simulate the obtained Petri net models through the Java based Petri net tool called PNetLab and analyse the SN P systems using simulation results. We also relate the languages generated by these variants of SN P systems and the corresponding Petri net models. The main contribution of the work presented is its potential for further development and exploitation of this structural link between SN P systems and Petri nets. For SN P systems this means that results, techniques and tools from the Petri net world become available, like linear algebra techniques (invariants) and coverability and reachability analysis including decidability results and verification techniques (model checking). Tools and techniques developed for Petri nets can be used for the description, analysis, and verification of behavioural properties of SN P system.

We also study the computational and generative power of spiking neural P systems with anti-spikes. We show that the idea of encoding 1 as spike and 0 as anti-spike proves to be very efficient in simulating Boolean circuits, finite state transducers, and solving NP-complete problems. We design SN PA systems simulating the operations of different Boolean gates. We also design SN P systems with anti-spikes to perform arithmetic operations like 2's complement, addition, and subtraction. The advantage of using this variant of SN P system is that we can perform the operations on negative numbers also. We show that any instance of SAT in conjunctive normal form, with n variables and m clauses is solved in a more efficient way using SN PA system.

1.4 An Overview of the Thesis

The thesis investigates the structural link between SN P systems and Petri nets by introducing algorithms to translate different variants of SN P systems into Petri nets. Thesis is organized as follows.

Chapter 2 gives mathematical definitions, notions, notations of formal languages and automata theory. It introduces P systems by presenting the definition of the basic model of P systems and by briefly describing some of the extensions of this model which can be found in the existing literature. It also presents preliminaries of SN P systems: description of different variants of SN P systems, computing with SN P systems, and the important properties of SN P systems. It illustrates the basic features of Petri nets with its languages. We conclude this chapter with a brief description of the Petri net tool called PNetLab.

In **Chapter 3** we investigate the language generative power of SN PA systems. It is shown that SN PA systems can generate languages that cannot be generated by SN P systems. We also study the power of SN PA systems as transducers. Boolean circuits and arithmetic operations have been simulated in this framework. At the end of the chapter we present a non-deterministic solution to SAT problem using SN PA systems.

We pass to the next major topic of the thesis in **Chapter 4**. The definition of Petri net with guard function which will be used in our translation is given. We show the similarities between SN P systems and Petri nets. We give a generalized algorithm to translate standard SN P systems into Petri nets. Some SN P systems are translated into Petri nets by using the algorithm. The obtained Petri net models are simulated using Java based Petri net tool called PNetLab. The simulation results are analysed. The results show that Petri net is an efficient tool to simulate, verify, and analyse the SN P systems. The language generated by the SN P systems are related with the step languages of the corresponding Petri nets.

In **Chapter 5** we translate SN P systems with anti-spikes into Petri nets. The annihilation rule $a\overline{a} \rightarrow \lambda$ that is implicitly present in each neuron of SN PA system is mapped to a pair of transitions. To simulate the execution of SN PA systems, each transition of the SN PA system is mapped to two consecutive steps of the Petri net. Again the SN PA systems are analysed using PNetLab. The language generated by the SN PA systems are also related with the step languages of the corresponding Petri nets.

In **Chapter 6** the relationship between the SN P systems and Petri nets is emphasized by modelling the simplex stop-and-wait protocol and producer/consumer problem using SN P systems. The models are translated into Petri nets using the algorithm proposed in Chapter 4. It is observed that there is a direct correspondence between the Petri net representation of the proposed models and standard solutions based on Petri nets already present in the literature.

Finally, in **Chapter 7** we draw some general conclusions and present suggestions for further research.

Chapter 2

Preliminaries

In this chapter we recall some basic concepts of formal language theory, P systems, and Petri nets, to the extent needed for describing the work presented in the thesis. Further information on these topics can be found in [37, 41, 43, 44, 54, 74, 93, 94].

2.1 Alphabets and Languages

An *alphabet* V is a finite nonempty set of symbols. The cardinality of the set V is denoted by $card(V)$. A string (or word) x over V is a sequence of symbols drawn from V. λ (or ϵ) denotes the empty string.

For an alphabet V, we denote by V^* (it is the set of all strings of symbols from V) the free monoid generated by V under the operation of *concatenation*. V^+ denotes the set of nonempty strings. i.e. $V^* - \{\lambda\}$.

Given a string $x \in V^*$ such that $x = x_1x_2$, for some $x_1, x_2 \in V^*$, then x_1, x_2 are respectively called a *prefix* and a *suffix* of x. If $x = x_1x_2x_3$, for some $x_1, x_2, x_3 \in V^*$, then x_2 is called a *substring* of x.

Any set of strings over an alphabet V, i.e., any subset of V^*, is called a *language*.

A language which does not contain the empty string is said to be λ-*free*. The usual set operations can be naturally extended to languages, as well as several operations which are specific for strings and sets of strings.

If L_1 and L_2 are two languages over V, then

- *Union* of L_1 and L_2 is the language that contains all strings that are either in L_1 or L_2.
  $L_1 \cup L_2 = \{x \mid x \in L_1 \; or \; x \in L_2\}$.

- *Intersection* of L_1 and L_2 is the language that contains all strings that are both in L_1 and L_2.
  $L_1 \cap L_2 = \{x \mid x \in L_1 \; and \; x \in L_2\}$.

- *Difference* of L_1 and L_2 is the language of all strings that are in L_1 and not in L_2.
  $L_1 \backslash L_2 = \{x \mid x \in L_1 \; and \; x \notin L_2\}$.
  When a particular alphabet V is understood from context, we shall write $\overline{L}$ - the complement of L - instead of the difference $V^* \backslash L$.

- *Complement* of a language L the set of all strings in V^* that are not in L
  $\overline{L} = \{x \mid x \in V^* \; and \; x \notin L\}$.

- *Concatenation* of L_1 and L_2 is the language of all strings formed by concatenating a string from L_1 with a string from L_2.
  $L_1 L_2 = \{xy \mid x \in L_1 \, and \, y \in L_2\}$.
  L^k means the set of all strings that can be obtained by concatenating k elements of L. i.e. L^k=k times concatenation of L.

- *Kleene star* of a language L, denoted by L^* is the set of all strings obtained by concatenating zero or more strings from L.
  That is $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \cdots$.

  $L^* = \bigcup_{i=0}^{\infty} L^i$, where
  $L^0 = \{\lambda\}$,

$L^1 = L$,

L^2=LL,

L^k=$L^{k-1}L$, and so on.

Positive closure of L is denoted as L^+ and is set of all strings obtained by concatenating one or more strings from L. So $L^+ = LL^*$=$\bigcup_{i=1}^{\infty} L^i$.

The *length* of a string $x \in V^*$ is denoted by $|x|$ and $|x|_a$ is the number of occurrences of the symbol $a \in V$ in the string x. For a language $L \subseteq V^*$, the set $length(L) = \{|x| \mid x \in L\}$ is called the *length set* of L. If FL is a family of languages. we denote by NFL the family of length sets of languages in FL.

A language $L \subseteq V^*$ is said to be *regular* if there is a *regular expression* E over V such that $L(E) = L$. Let $L(E)$ be the language that regular expression E represents. A recursive definition for E (and $L(E)$) is given below:

- **Basis**: λ and $\emptyset$ are regular expressions, and $L(\lambda) = \{\lambda\}$ and $L(\emptyset) = \emptyset$. For any $a \in V$, a is a regular expression and $L(a) = \{a\}$.
- **Induction**: If E_1 and E_2 are regular expressions, then $E_1 + E_2$ is a regular expression, with $L(E_1 + E_2) = L(E_1) \cup L(E_2)$, and E_1E_2 is a regular expression, with $L(E_1E_2) = L(E_1)L(E_2)$. If E is a regular expression, then E^* is a regular expression, with $L(E^*) = (L(E))^*$, and (E) is a regular expression, with $L((E)) = L(E)$.

When $V = \{a\}$ is a singleton set, then the regular expression a^* denotes the set of all strings formed using a. i.e the set $\{\lambda, a, a^2, a^3, ...\}$ and $a^+ = aa^*$.

For $V = \{a_1, a_2, \ldots, a_n\}$ and $x \in V^*$, the *Parikh vector* associated with word x with respect to alphabet V is defined as $\Psi_V(x) = \langle |x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n} \rangle$. The *Parikh set* of a language $L \subseteq V^*$ is $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$. If V is a singleton set then $\Psi_V(L) = \{|x| \mid x \in L\}$.

The set of non negative integers and natural numbers are denoted by $\mathbb{N}$ and $\mathbb{N}^+$ respectively. A set Q of vectors in $\mathbb{N}^{+n}$, for some $n \geq 1$, is said to be *linear* if there are some vectors $v_1, v_2, \ldots, v_m \in \mathbb{N}^{+n}$, $m \geq 0$, such that $Q = \{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_i, \alpha_2, \ldots, \alpha_n \in \mathbb{N}^+\}$. A finite union of linear sets is a *semilinear* set. A language $L \subseteq V^*$ is *semilinear* if its Parikh image $\Psi_V(L)$ is a semilinear set.

A *multiset* over a set V is a function $M : V \to \{0, 1, 2, \ldots\}$. For each $a \in V$, $M(a)$ is the number of copies of a in the multiset M. The *cardinality* of M is $|M| = \sum_{a \in V} M(a)$. For two multisets M_1 and M_2 over V, the sum $M_1 + M_2$ is the multiset given by the formula $(M_1 + M_2)(a) = M_1(a) + M_2(a)$ for all $a \in V$, and if $k \in \mathbb{N}^+$ then $k.M_1$ is the multiset given by $(k.M_1)(x) = k.M_1(a)$ for all $a \in V$. We denote $M_2 \leq M_1$ whenever $M_2(a) \leq M_1(a)$ for all $a \in V$, and if $M_2 \leq M_1$, then the difference $M_1 - M_2$ is $M_1(a) - M_2(a)$ for all $a \in V$. The empty multiset is denoted by λ.

A multiset over V can be naturally represented as a string of elements from V. For instance a string $x = a\overline{a}aa\overline{a}$ denotes a multiset M over $V = \{a, \overline{a}\}$ with $M(a) = 3$, and $M(\overline{a}) = 2$. Clearly all permutations of the string x represent same multiset M.

With $P(V^*)$ we denote the set of *families of languages* over an alphabet V. If we consider two alphabets U and V, a mapping $h : V \to P(U^*)$, extended to $h : V^* \to P(U^*)$ by $h(\lambda) = \lambda$ and $h(x_1x_2) = h(x_1)h(x_2)$, for $x_1, x_2 \in V^*$, is called a *substitution*. For a language $L \in V^*$, we have $h(L) = \bigcup_{x \in L} h(x)$.

If the set $h(a)$ is finite for each $a \in V$, then h is called a *finite substitution*; if $card(h(a)) = 1$, then h is called a *morphism*.

If $\lambda \notin h(a)$, for each $a \in V$, then h is said to be a *λ-free substitution* (or *λ-free morphism*). For a morphism $h : V^* \to U^*$, the *inverse morphism* can be obtained defining a mapping $h^{-1} : U^* \to P(V^*)$ by $h^{-1}(y) = \{x \in V^* \mid h(x) = y\}$.

A morphism $h : V^* \to U^*$ is called a *coding* if $h(a) \in U$ for all $a \in V$, it is called a *weak coding* if $h(a) \in U \cup \{\lambda\}$ for each $a \in V$.

The mappings defined on strings are extended to languages in the natural way, for instance if $h : V^* \to U^*$ is a morphism and $L \subseteq V^*$ then $h(L) = \{h(x) \mid x \in L\}$.

2.2 Automata

An automaton is an abstract model of a digital computer. It has a mechanism for reading input. It is assumed that the input is a string over a given alphabet, written on an *input tape*, which the automaton can read but not change. The input tape is divided into cells, each of which can hold one symbol. The input mechanism can read the input tape left to right, one symbol at a time. It can also detect the end of the input string. The automaton can produce output of some form. It may have a temporary *storage* device which consists of an unlimited number of cells, each capable of holding a single symbol from an alphabet (not necessarily the same one as the input alphabet). The automaton can read and change the contents of the storage cells. Finally, the automaton has a *control unit* which can be any one of a finite number of *states*.

An automaton is assumed to operate in a discrete time frame. At any given time, the control unit is in some state and the input mechanism is scanning a particular symbol on the input tape. The state of the control at the next time step is determined by a *transition function*. A transition function gives the next state in terms of the current state, the current input symbol, and the information currently available in the temporary storage. During a transition from one time interval to another, output may be produced or the information in the temporary storage may be changed.

An automaton whose output response is limited to a simple *yes* or *no* is called an *acceptor* or *recognizer*. A more general automaton, capable of producing strings of symbols as output is called a *transducer*.

Based on the nature of the temporary storage, we have three types of automata,

namely, *finite automata*, *pushdown automata*, and *Turing machines*. Here we only discuss about finite automata and Turing machines.

2.2.1 Finite Automata

Finite automata is characterized by having no temporary storage. So, a finite amount of information can be retained in the control unit by placing the unit into specific state. Since the number of such states is finite, a finite automaton can only deal with situations in which the information to be stored at any time is strictly bounded.

Definition 2.1 (**Finite Automaton**). *A finite automaton is a construct*

$$M = (Q, V, \delta, q_0, F),$$

where Q and V are disjoint alphabets, $q_0 \in Q$, $F \subset Q$, and $\delta : Q \times (V \cup \lambda) \to 2^Q$; Q is a finite set of states, V is a finite set of input alphabet, q_0 is an initial state, F is a set of final states, and δ is a transition function.

If $card(\delta(q, a)) \leq 1$ for all $q \in Q$, and $a \in V$, we say that the automaton is *deterministic*.
We can extend the definition of a transition function from δ to δ^* as follows:

$$\delta^* : Q \times V^* \to 2^Q,$$
$$\delta^*(q, \lambda) = \{q\},$$
$$\delta^*(q, xa) = \{p \mid \text{for some state } r \text{ in } \delta^*(q, x), p \text{ is in } \delta(r, a)\},$$

where $q \in Q$, $x \in V^*$, and $a \in V$.

The language accepted by a finite automaton M is defined as

$L(M) = \{w \in V^* \mid \delta^*(q_0, w) \in F\}$.

It is known that both deterministic and nondeterministic finite automata characterize the same family of languages, namely, the family of *regular* languages.

A *deterministic transducer* $M' = (Q, V, \Delta, \delta, \mu, q_0, F)$ is defined as a finite automaton with output associated with its moves. The components Q, V, δ, q_0, F are same as in deterministic finite automaton, Δ is the *output alphabet* and μ is the output function that maps $Q \times V \to \Delta$, i.e. a symbol is also produced when reading a symbol in the input string.

2.2.2 Turing Machines

Turing machine is characterized by having a single, one-dimensional array of cells, each of which can hold a single symbol. This array extends indefinitely in both directions and is therefore capable of holding an unlimited amount of information. The information can be read and changed in any order. We call such a storage device a *tape*. Associated with the tape is a *read-write* head that can move right or left on the tape. On each move, the read-write head can read and write a single symbol on the tape.

Definition 2.2 (**Turing Machine**). *A Turing machine M is a construct*

$$M = (Q, V, \Gamma, \delta, q_0, \square, F),$$

where Q, V, and Γ are disjoint alphabet, $q_0 \in Q$, $\square \in \Gamma$, $F \subseteq Q$, and $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$; Q is a finite set of states, V is a finite set of input alphabet, Γ is finite set of tape alphabet, q_0 is an initial state, $\square$ is a blank symbol, F is a set of finite states, and δ is a transition function.

We assume that $V \subseteq \Gamma - \{\square\}$. If $(p, b, D) \in \delta(q, a)$ for $q, p \in Q$, $a, b \in \Gamma$, and $D \in \{L, R\}$, the machine reads a symbol a in a state q and passes to a state p, replaces the symbol a with a symbol b and then moves the read-write head with respect to D.

To exhibit the configuration of a Turing machine, we use the idea of an instantaneous description. Any configuration is completely determined by the current state of the control unit, the contents of the tape, and the position of the read-write head. We use the notation in which x_1qx_2 is the instantaneous description of a machine in state q with x_1 as the tape content on the left side of the read-write head and x_2 as the tape content on the right side of the read-write head. The read-write head is reading the first symbol of x_2.

A move from one configuration to another is defined by $\vdash$, Thus,

$$x_1qax_2 \vdash_M x_1bpx_2 \text{ iff } (p, b, R) \in \delta(q, a);$$
$$x_1cqax_2 \vdash_M x_1pcbx_2 \text{ iff } (p, b, L) \in \delta(q, a);$$
$$x_1q\Box \vdash_M x_1ap \text{ iff } (p, a, R) \in \delta(q, \Box);$$
$$x_1bq\Box \vdash_M x_1pba \text{ iff } (p, a, L) \in \delta(q, \Box);$$

where $x_1, x_2 \in (\Gamma - \{\Box\})^*$; $a, b, c \in \Gamma$, and $q, p \in Q$.

Always the automaton starts in the given initial state with some information on the tape. It then goes through a sequence of steps controlled by the transition function δ. During this process, the contents of any cell on the tape may be examined and changed many times. Eventually, the whole process may terminate, which we achieve in a Turing machine by putting it into a *halt state*. A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined. But we assume that no transitions are defined for any final state, so a Turing machine will halt whenever it enters a final state.

The language of strings accepted by a Turing machine M is defined as

$L(M) = \{w \in V^* \mid q_0w \overset{*}{\vdash}_M x_1q_fx_2 \text{ for some } q_f \in F, \text{ and } x_1, x_2 \in \Gamma^*\}$.

When w is not in $L(M)$, one of two things can happen; the machine can halt in a non-final state or it can enter an infinite loop and never halt.

A *deterministic* Turing machine is an automaton for which $card(\delta(q,a)) \leq 1$ for all $q \in Q$ and $a \in \Gamma$.

The class of deterministic Turing machines are equivalent to the class of non-deterministic Turing machines and the family of languages accepted is exactly the family of *recursively enumerable* languages. The family of Turing computable sets of numbers is denoted by NRE (these sets are length sets of RE languages, hence the notation).

We can use Turing machines not only as language acceptors but also as transducers. The string of non-blank symbols present on the tape at the initial configuration is called the input. The string of non-blank symbols present on the tape at the end of the computation is called the output. Thus, we can view a Turing machine as a transducer M implementing a function f defined by

$$w' = f(w),$$

provided that
$q_0 w \vdash^*_M w_1 q_f w_2$, where $w' = w_1 w_2$, for some final state q_f.

A function $f : V^* \rightarrow \Gamma^*$ is said to be *computable*, if there exists some Turing machine $M = (Q, V, \Gamma, \delta, q_0, \Box, F)$ such that starting with w as the non-blank portion on the input tape, we end up in $f(w)$ as the non-blank portion at the end of the computation.

Definition 2.3 (**Universal Turing Machine**)**.** *A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w, can simulate the computation of M on w.*

A computer is a programmable machine, able to execute any program it receives. The computing power of Turing machines can be known from the well known *Church-Turing thesis*.

Church-Turing Thesis: Any computation that can be carried out by mechanical means can be performed by some Turing machine.

2.3 Chomsky Grammars

A *grammar* is a device which generates all the strings of a language. To formally define a grammar, we start from the notion of *rewriting systems*, which is a pair $\gamma = (V, P)$, where V is an alphabet and P is a set of productions or rewriting rules, and usually they are written in the form $u \to v$. For $x, y \in V^*$, we write

$x \Rightarrow_\gamma y$ iff $x = x_1 u x_2$ and $y = x_1 v x_2$, for some $x_1, x_2 \in V^*$, and $u \to v \in P$.

When γ is understood we write $\Rightarrow$ instead of $\Rightarrow_\gamma$. The *reflexive transitive closure* of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$.

Definition 2.4 (**Chomsky Grammar**). *A Chomsky grammar is a quadruple $G = (N, T, S, P)$, where N and T are disjoint alphabets, $S \in N$, and P is a finite subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$; N is a finite set of non-terminal alphabet, T is finite set of terminal alphabet, S is the start symbol (or axiom), and P is a finite set of rules of G. The rules of P are written in the form $u \to v$ with $|u|_N \geq 1$.*

The language generated by G is defined as

$$L(G) = \{x \in T^* \mid S \overset{*}{\Rightarrow} x\}.$$

In other words, $L(G)$ consists of all the string of terminal symbols which can be derived from the axiom S. Each string $w \in (N \cup T)^*$ such that $S \overset{*}{\Rightarrow} w$ is called a *sentential form*.

Two grammars G_1 and G_2 are said to be equivalent, if $L(G_1) - \{\lambda\} = L(G_2) - \{\lambda\}$.

Generally, we consider two grammars equivalent if they generate the same language when we ignore the empty string.

According to the form of their rules, Chomsky grammars are classified as follows.
A grammar $G = (N, T, S, P)$ is called

- *length-increasing*, if for all $u \to v \in P$ we have $|u| \leq |v|$;
- *context-sensitive*, if each $u \to v \in P$ has $u = u_1 A u_2$ and $v = u_1 x u_2$, for u_1 and $u_2 \in (N \cup T)^*$, $A \in N$, and $x \in (N \cup T)^+$; (In length-increasing and context-sensitive grammars the rule $S \to \lambda$ is allowed, provided S does not appear in the right-hand members of rules in P.)
- *context-free*, if each rule $u \to v \in P$ has $u \in N$;
- *linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup T^* N T^*$;
- *right-linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup T^* N$;
- *left-linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup N T^*$;
- *regular*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

The *arbitrary*, *length-increasing*, *context-free*, and *regular grammars* are also called *type 0*, *type 1*, *type 2*, and *type 3* grammars, respectively.

The family of languages generated by length-increasing is equal to the family of languages generated by context-sensitive grammars; the family of languages generated right- or by left-linear grammars coincide and they are equal to the family of languages generated by regular grammars.

We denote by REG, LIN, $SLIN$, CF, CS, and RE (*recursively enumerable*) the families of languages generated by regular, linear, semilinear, context-free, context-sensitive, and arbitrary grammars, respectively. We denote by FIN the family of finite languages. $SLIN$ does not belong to Chomsky hierarchy. It is the family of languages whose Parikh mapping is semilinear.

By $NFIN$, $NSLIN$, NRE we denote the families of finite, semilinear, and Turing computable sets of (positive) natural numbers (number 0 is ignored); they correspond to the length sets of finite, semilinear, and recursively enumerable languages, whose families are denoted by FIN, $SLIN$, RE.

Following theorem gives the relationship between different families of languages:

Theorem 2.1. *Chomsky hierarchy:* $FIN \subset REG \subset LIN \subset CF \subset CS \subset RE$.

The importance of the role of Chomsky hierarchy is owing to several reasons: the family of RE languages generated by type-0 Chomsky grammars is exactly the family of languages which are recognized by Turing machines, and according to the Turing-Church thesis this is the maximal level of algorithmic computability; the Chomsky hierarchy is well structured, hence we have a detailed classification of computing machineries.

We denote by $BFIN$, $BREG$, BCF, BCS, BRE the families of finite, regular, context-free, context-sensitive, and recursively enumerable languages over the binary alphabet $B = \{0, 1\}$.

Example 2.1. *A list of languages in the Chomsky hierarchy:*
$L_1 = \{0^m 1^n \mid m, n \geq 1\} \in REG$;
$L_2 = \{0^n 1^n \mid n \geq 1\} \in CF \backslash REG$;
$L_3 = \{0^n 1^n 0^n \mid n \geq 1\} \in CS \backslash CF$;
$L_4 = \{0^{2^n} \mid n \geq 1\} \in CS \backslash CF$

Normal Forms

We present here three classical normal-form theorems for grammars. The first two theorems concern context free grammars, while the third one is a normal form for type-0 grammars.

Theorem 2.2 (Chomsky Normal Form)**.** *Any λ-free context-free language can be generated by a grammar in which all productions are of the form $A \to BC$ or $A \to a$, where A, B and C are non-terminal symbols, and a is a terminal symbol.*

Theorem 2.3 (Greibach Normal Form)**.** *Any λ-free context-free language can be generated by a grammar in which all productions are of the form $A \to a\alpha$, where A is a non-terminal symbol, a is a terminal symbol, and α is a (possibly empty) string of non-terminal symbols.*

Theorem 2.4 (Kuroda Normal Form)**.** *Any type-0 Chomsky grammar can be generated by a grammar in which all productions are context-free rewriting rules, or they are of the form $AB \to CD$, where A, B, C, and D are non-terminal symbols.*

2.4 Register Machines

A very useful characterization of NRE (the family of sets of numbers which are Turing computable) is obtained from register machines (also called counter machines, program machines) [74].

A *non-deterministic* register machine is a construct $M = (m, H, l_0, l_h, I)$; where m is the number of registers, H is the set of instruction labels, l_0 is the start label (labelling an ADD instruction), l_h is the halt label (assigned to instruction $HALT$), and I is the set of instructions; each label from H labels only one instruction from I, thus precisely identifying it. The instructions are of the following forms:

- $l_i : (ADD(r), l_j, l_k)$ (add 1 to register r and then go to one of the instructions with labels l_j,l_k non-deterministically chosen),
- $l_i : (SUB(r), l_j, l_k)$ (if register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k),
- $l_h : HALT$ (the halt instruction).

A register machine M generates a set $N(M)$ of numbers in the following way: we start with all registers empty (i.e., storing the number zero), we apply the instruction with label l_0 and we continue to apply instructions as indicated by the labels (and made possible by the contents of registers); if we reach the halt instruction, then the number n present in the first register at that time is said to be generated by M. (Without loss of generality we may assume that in the halting configuration all other registers are empty; also, we may assume that register 1 is never subject of SUB instructions, but only of ADD instructions.)

A register machine can also be used as a number accepting device: we introduce a number n in some register r_0, we start working with the instruction with label l_0, and if the machine eventually halts, then n is accepted. Again, accepting register machines characterize NRE.

In both the accepting and the computing case, the register machine can be deterministic, i.e., with the ADD instructions of the form $l_i : (ADD(r), l_j)$ (add 1 to register r and then go to the instruction with label l_j). Again, without loss of generality, we may assume that all registers are empty in the halting configuration.

A register machine can also be used for defining a language, in the following way.

Let $V = \{a_1, a_2, \ldots, a_s\}$, for some $s \geq 1$. For a string $x \in V^*$, let us denote by $val_s(x)$, the value in base $s+1$ of x. (We use base $s+1$ in order to consider the symbols of $a_1, a_2, \ldots, a_s$ as digits $1, 2, \ldots s$, thus avoiding the digit 0 in the left hand of the string.) We extend this notation in the natural way to the set of strings.

Proposition 2.1. *If $L \subseteq V^*$, $card(V) = m$, $L \in RE$, then a 3 register machine M exists such that for every $x \in V^*$ we have $x \in L$ if and only if M halts when starting with $val_{s+1}(x)$ in its first register, in the halting step, all registers of the machine are empty, [74].*

2.5 Boolean Functions and Circuits

A Boolean function is a function f from the Cartesian product $\{0,1\}^n$ to $\{0,1\}$. Alternatively, we write $f : \{0,1\}^n \to \{0,1\}$. The set $\{0,1\}^n$, by definition, the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ where each x_i is either 0 or 1, is called the domain of f. The set $B = \{0,1\}$ is called the co-domain of f. Because Boolean functions are related to logic, we think of 0 as "false" and 1 as "true".

There are three primary Boolean functions that are widely used: The NOT function - this is just a negation; the output is the opposite of the input. The NOT function takes only one input, so it is called a unary function or operator. The output is true when the input is false, and vice-versa. The AND function - AND function returns true only if all inputs are true; if there is an input which is false the function returns false. The OR function - the output of an OR function is true if at least one of its inputs is true.

Apart from these three primary Boolean functions, there are two universal functions: The NAND or Not AND function - this is a combination of two separate logical functions, the AND function and the NOT function connected together in series. The NAND Function returns true only when any of its inputs are false. NOR or Not OR function - this is also a combination of two separate functions, the OR function and the NOT function connected together in series. The NOR function outputs true only when all of its inputs are false. The NAND and NOR functions are universal functions because they are sufficient to implement any Boolean function and can be combined to form any other functions like OR, NOT and AND. Except NOT function, all other Boolean functions can have any number of inputs, with a minimum of two.

Every n-ary Boolean function can be expressed as a Boolean expression in n variables $X = x_1, x_2, \ldots, x_n$ and two expressions are logically equivalent if and only if they express the same Boolean function.

Boolean functions may be practically implemented by using electronic gates. Any logic function can be implemented using either primary Boolean gates or only NAND gates or only NOR gates. To implement using NAND gates, first the logic function has to be written in sum of product (SOP) form. Once logic function is converted to SOP, then it is very easy to implement using NAND gates. Similarly any logic function in product of sums form can be implemented using NOR gates.

There is a potentially more economical way than expressions for representing Boolean functions- namely Boolean circuits. A Boolean circuit is a graph $C = (V, E)$, where the nodes in $V = \{1, \ldots, n\}$ are called the *gates* of C. The graph C has a rather special structure. First, there are no cycles in it, so we can assume that all edges are of the form (i, j), where $i < j$. All nodes in the graph have the "in-degree" (number of incoming edges) greater than or equal to 1. Also, each gate $i \in V$ has a *sort* $s(i)$ associated with it, where $s(i) \in \{true, false, \vee, \wedge, \neg, \overline{\vee}, \overline{\wedge}\} \cup \{x_1, x_2, \ldots\}$. If $s(i) \in \{true, false\} \cup \{x_1, x_2, \ldots\}$, then the in degree of i is 0, that is, i must have no incoming edges. Gates with no incoming edges are called the inputs of C. If $s(i) = \neg$, then i has "in-degree" one. If $s(i) \in \{\vee, \wedge, \overline{\vee}, \overline{\wedge}\}$, then the "in-degree" of i must be greater than or equal to two. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges) is called the output gate of the circuit.

This concludes our definition of the syntax of circuits. The semantics of circuits specifies a truth value for each appropriate truth assignment. We let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X | s(i) = x$ for some gate i of $C\}$). We say that a truth assignment T is appropriate for C if it is defined for all variables in $X(C)$. Given such a T, the truth value of gate $i \in V$, $T(i)$ is defined, by induction on i, as follows: If $s(i) =$ true then $T(i) =$ true, and, similarly, if $s(i) =$ false, then $T(i) =$ false. If $s(i) \in X$, then $T(i) = T(s(i))$. If now $s(i) = \neg$, there is a unique gate $j < i$ such that $(j, i) \in E$. By induction, we know $T(j)$, and then $T(i)$ is true if $T(j) =$ false, and vice-versa. Let $C_i = \{j \mid (j, i)$ is an edge entering $i\}$. If

$s(i) = \vee$, then $T(i)$ is true if only if at least for one of $j \in C_i$, $T(j)$ is true. If $s(i) = \wedge$, then $T(i)$ is true iff for all $j \in C_i$, $T(j)$ is true. If $s(i) = \overline{\vee}$ (symbol for NOR), then $T(i)$ is true if only if for all $j \in C_i$, $T(j)$ is false. If $s(i) = \overline{\wedge}$ (symbol for NAND), then $T(i)$ is true iff at least for one of $j \in C_i$, $T(j)$ is false. Finally, the value of the circuit, $T(C)$, is $T(n)$, where n is the output gate.

2.6 P systems

A P system is a computing model which abstracts from the way the alive cells process chemical compounds in their compartmental structure. In short, we have a membrane structure, consisting of several membranes embedded in a *main membrane* (called the skin) and delimiting regions where multisets of certain objects are placed.

A membrane structure is represented by a Venn diagram as shown in the Figure 2.1 and is identified by a string of correctly matching parentheses, with a unique external pair of parentheses; this external pair of parentheses corresponds to the external membrane, called the *skin*. The membrane structure corresponding to the Venn diagram is $[_1[_2]_2[_3]_3[_4[_5[_7]_7[_8]_8]_5[_6]_6]_4]_1$.

A membrane without any other membrane inside is said to be *elementary*. We say that the number of membranes is the *degree* of the membrane structure, while the height of the tree associated in the usual way with the structure is its *depth*. The membranes delimit regions (each region is bounded by a membrane and the immediately lower membranes, if there are any).

In each region, we have objects which evolve according to given rules (*evolution rules* or *symport/antiport rules*). The objects can be described by symbols or by strings of symbols (in the former case their multiplicity matters, that is, we work

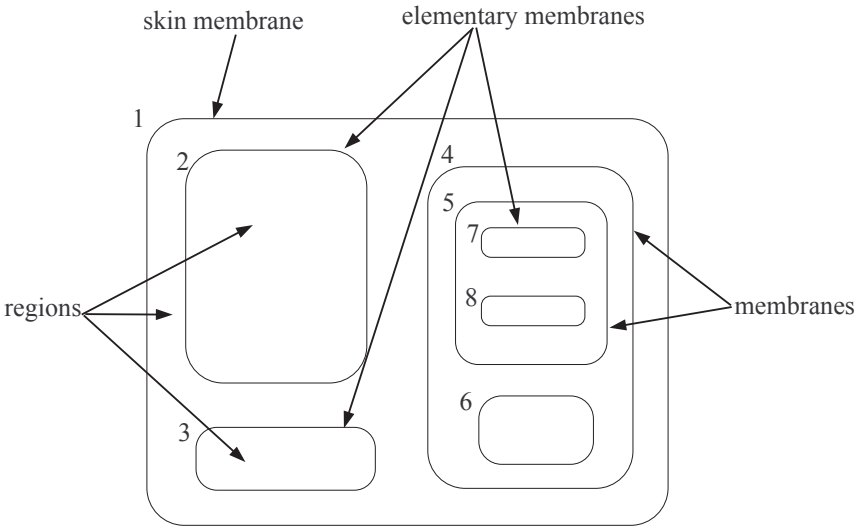


Figure 2.1: Structure of a P system

with multisets of objects placed in the regions of the membrane structure; in the second case we can work with languages of strings or, again, with multisets of strings). 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 rules are applied non-deterministically (the rules to be used and the objects to evolve are randomly chosen) in a maximally parallel manner (in each step, all objects which can evolve must do it). The objects can also be communicated from a region to another one. In this way, we get transitions from a configuration of the system to the next one.

A sequence of transitions constitutes a computation; with each halting computation we associate a result, in the form of the objects present in a given membrane in the halting configuration, or expelled from the system during the computation.

2.6.1 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.

Definition 2.5 (**P System with symbol objects**). *A P system of degree* $m \geq 1$ *with symbol-objects is a tuple:*

$$\Pi = (O, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0) \textit{ where}$$

- O *is an alphabet and its elements are called objects;*
- μ *is a membrane structure of degree* m*; with the membranes (and hence the regions that they delimit) are labelled in a one-to-one manner with the elements from the set* $1, 2, \ldots, m$*;*
- $w_1, w_2, \ldots, w_m \in V^*$ *are the multiset of objects associated with the regions of* μ*;*
- R_i, $1 \leq i \leq m$, *are the finite sets of evolution rules over* O*;* R_i *is associated with the region* i *of* μ*; an evolution rule is of the form* $u \to v$*, where* $u \in O^+$ *and* v *is a string over* $\{a_{here}, a_{out} | a \in O\} \cup \{a_{in_j} | a \in O\, 1 \leq j \leq m\}$*;*
- $i_0 \in \{0, 1, 2, \ldots, m\}$*; if* $i_0 \in \{1, \ldots, m\}$*, then it is the label of an elementary membrane that encloses the output region; if* $i_0 = 0$*, then the output region is the environment.*

For any evolution rule $u \to v$, the length of u is called the *radius* of the rule and the symbols $here$, out, in_j , $1 \leq j \leq m$, are called *target* indications.

To simplify the notation the target indication $here$ is omitted. According to the size of the radius of the evolution rules we distinguish between *cooperative systems* (if the radius is greater than one) and *non-cooperative systems* (otherwise).

A special class of cooperative systems is that of *catalytic systems*, where a subset $C \subseteq O$ of special symbol-objects (called *catalysts*) is fixed. In case of catalytic systems, the system is of the form

$$\Pi = (O, C, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0).$$

In such systems the evolution rules can be of two different kinds: $a \to v$ and $ca \to cv$ (catalytic rules) where $c \in C$, $a \in O - C$, and v is a string over $\{a_{here}, a_{out} | a \in O - C\} \cup \{a_{in_j} \mid a \in O - C, 1 \leq j \leq m\}$.

The *initial configuration* of the system Π is constituted by the membrane structure μ and the multisets represented by the strings $w_i, 1 \leq i \leq m$. In general, we call *configuration* of the system the membrane structure and the tuple of multisets of objects present in the regions of the system.

A transition between configurations is executed using the evolution rules in a *non-deterministic maximally parallel manner* (we suppose that a global clock exists, marking the time for the whole system). This means that the objects are assigned to the rules in such a way that, after this assignation, no other rules can be applied to the objects that have been not assigned and this procedure is applied in parallel in each region of the system, at each step. If an object can be used by several evolution rules, then the choice is made in a non-deterministic way.

The application of an evolution rule $u \to v$ in a region i means to remove the multiset of objects identified by u from region i, and to add the objects specified by the multiset v, in the regions specified by the target indications associated to each object in v. In particular, if v contains an object a with target indication $here$, then the object a will be placed in the region i where the evolution rule has been applied. If v contains an object a with target indication out, then the object a will be moved to the region immediately outside the region i (this can be the environment if the region where the rule has been applied is the *skin* membrane). If v contains an object a with target indication in_j then the object a is moved from the region i and placed

into the region j (this can be done only if such region j is immediately inside region i; otherwise the evolution rule $u \rightarrow v$ cannot be applied).

It is also possible to use target indications of the forms $here$, out, in, with in being a weaker version of in_j; in such a case, an object a having the target indication in goes to any lower level membrane, non-deterministically chosen (if no inner membrane exists, then the rule cannot be applied).

In other words, by using the evolution rule $u \rightarrow v$ in a region i that contains a multiset identified by w, we subtract the multiset identified by u from the multiset identified by w, and then we add the objects specified by v to the multiset of the region i and to the multisets of adjacent regions according to the target indications specified. In this way, at each step, all the multisets associated to the regions of the system evolve and the system passes from some configuration to a new one; this passage is called *transition*.

A sequence of transitions between configurations of a system Π is called a *computation*; a computation is *successful* (or *halting*) if and only if it *halts* and this means that there is no rule applicable to the objects present in the last configuration.

P systems can be used as acceptors or generators of sets of numbers. P system generators start with a fixed initial configuration and non-deterministically compute. The output is defined as the number of objects present in the output membrane in the halting configuration of Π. In an acceptor, an input is given in a specified membrane(s) before the computation begins. If the computation halts, the input is accepted. (For additional general definition details see [35].)

Many different classes of P systems have been investigated, and most of them turned out to be computationally complete with respect to the Turing-Church thesis (i.e., equal in power to Turing machines).

Rewriting rules discussed above were associated with (located in) regions and

were acting on multisets of objects residing in these regions. Another type of rule is associated with (located on) membranes, and they govern the movement/exchange of objects (located in neighbouring regions) through these membranes. These types of rules are referred to as communication rules, because exchange of objects between neighbouring regions is the way that regions "communicate" with each other. Symport and antiport rules are typical examples of communication rules [88]. Many results on this kind of P systems can be found in [24, 61] and about their computational power using Petri nets can be found in [25]. A P system with communication rules is defined in a similar fashion to a P system with multiset rewriting rules, but with some essential differences. First of all, we consider now communication rules rather than rewriting rules - sets of such rules are associated with membranes rather than with regions. Moreover, since there are communication rules only, one needs a supply of objects that can be moved into the system. Otherwise, the multiset of objects present in the system will never change and consecutive configurations of a computation would present merely distributions of this fixed multiset through the regions of the system.

Besides the two main classes of P systems, with multiset rewriting rules and communication rules, several other classes are considered in the literature. The maximal parallelism is the mostly used regime of applying the rules in a P system, but there are other possible modes of applications of rules. For instance, one can use any multiset of (jointly) applicable rules in a region (the asynchronous mode), or only one applicable rule in the whole system (the sequential mode), or a specified number of rules in the whole system or in each compartment (bounded parallelism), etc.

A possible weakness of P systems considered in this chapter so far, especially from a biological point of view, is the fact that the membrane structure is static, it does not evolve during the computation. In some models, rules have additional capabilities and can trigger the membranes themselves to divide, dissolve, or create

new membranes. For example, the model of P Systems with active membranes allows these additional capabilities [29, 82]. Another type of P systems (called P systems with active objects) introduced in [70] allows rules to create new membranes during the computation. The objects of an active P system consist of passive objects which do not create new membranes and active objects which do create new membranes.

In P systems with symbolic objects, the objects were considered atomic, in the etymological sense, and were given by symbols from a given alphabet. However, it is natural to consider also P systems whose objects are structured. There are many natural motivations to consider such structured objects-they come from computer science, mathematics, and biology. For instance, many molecules such as proteins, DNA, RNA have string-like structures. Hence one can describe such objects by strings over a given alphabet. For P systems operating on string-like objects we have to specify the alphabet, the membrane structure, the strings present in the system at the beginning of the computation, the rules for processing the strings, and the way to define the result of a computation. The strings of a P system can be considered either as a set (languages) or as a multiset. Then, the rules must be string processing rules one can consider, among others, parallel string rewriting such as in Lindenmayer systems, or sequential string rewriting such as in Chomsky grammars. Another fruitful idea is to use string evolving rules which are specific for DNA processing, such as splicing or insertion-deletion. To generate an exponential workspace (for "solving" computationally hard problems) one can use operations which replicate strings.

Many of these variants lead to computationally universal systems, while several variants with an enhanced parallelism are able to solve NP-complete problems in polynomial (often, linear) time - of course, by making use of an exponential space. A series of applications, in biology, linguistics, computer science, management, etc., were reported.

Until now we have considered hierarchical arrangement of membranes which correspond to cell-like membrane structure described by trees. Tissue P systems have been motivated by the structure and behaviour of multicellular organisms where they form a multitude of different tissues performing various functions [71]; the structure of the system is instead represented as a graph where nodes are associated with the cells which are allowed to communicate alongside the edges of the graph.

Combining the ideas of tissue P systems and spiking neurons, spiking neural P systems (SN P systems for short) were introduced.

2.6.2 Spiking Neurons

Information processing in the brain depends not only on the properties of neural networks but also on the properties of processing units-neurons. We recall here from [50, 67] some notions about the neural cell, mainly focusing on the electric pulses a neuron is transmitting through its synapses; such a pulse is usually called a spike, or action potential which was the major inspiration for the development of spiking neural P systems.

Neuron, a one-membrane cell, is the basic element in the human brain. Figure 2.2 shows schematic illustration of a neuron and its parts. It consists of synapses, a soma, dendrites, an axon, an axon hillock and axon terminals. Neurons use short and sudden increase in voltage to send information. A neuron receives small electrical pulses usually called spikes from other neurons through its dendrites. The spikes are stored and processed in the cell body called soma. Dendrites are thin numerous bushy extensions of the cell.

They receive spikes from the synapses and carry them to soma. When the weight (number of spikes) gets bigger than a particular value, associated with axon hillock, called the threshold, a neuron fires; that is, it emits an output signal called a

spike. Spikes are produced in the initial segment of an axon (the only neuronal output extension). Then they quickly propagate along the axon towards other neurons

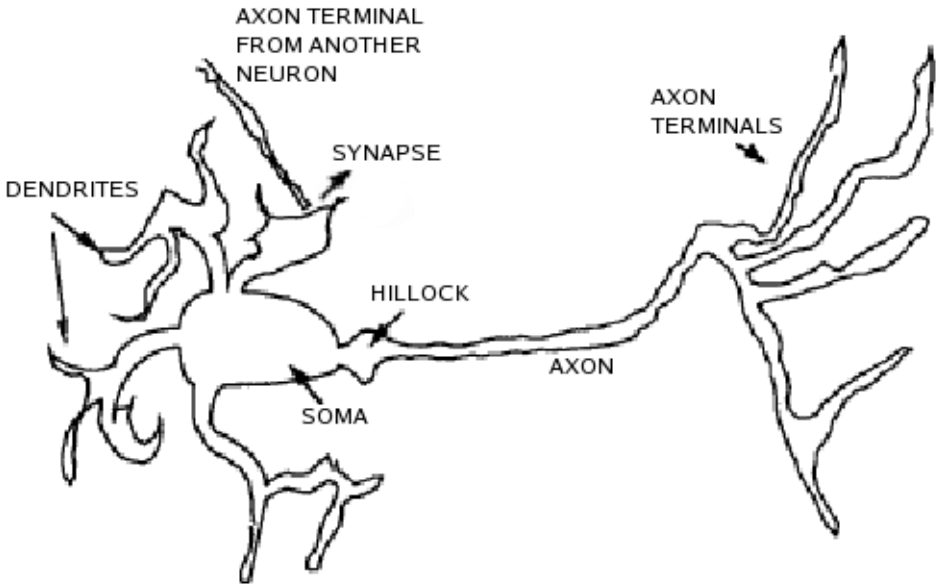


Figure 2.2: Schematic illustration of a neuron and its parts.

within a network. At its distant end an axon makes thousands of branches each of which is called axon terminal, which is the input to other neuron. Spikes cannot just cross the gap between one neuron and the other. They have to be handled by the most complicated part of the neuron: the synapse, connection between the axon terminal of a neuron and the dendrite of the other neuron. Neurons send out erratic sequences of spikes, or spike-train, which alter dramatically in frequency over a short period of time. Since all spikes of a given neuron look alike, the form of the action potential does not carry any information. Rather, it is the number and the timing of spikes what matter.

So, the size and the shape of a spike is independent of the input of the neuron, but the time when a neuron fires depends on its input. Action potentials in a spike train are usually well separated. Even with very strong input, it is impossible to excite a second spike during or immediately after a first one. The minimal distance

between two spikes defines the refractory period of the neuron. Neurons use this spatial and temporal information of incoming spike patterns to encode their message to other neurons. A typical neuron is able to receive thousands of spikes and transmit thousands of spikes concurrently. There are many different schemes to use this spike timing information in neural computation.

In the next section, we will capture some of these ideas in the framework of neural-like P systems as existing in the membrane computing literature, adapting the definition to the case of spiking.

2.6.3 Neural-Like P Systems

We also recall here the initial definition of a neural-like P system as considered in [9, 50]. The basic idea is to consider cells related by synapses and behaving according to their states; the states can model the firing of neurons, depending on the inputs, on the time of the previous firing, etc.

Definition 2.6 (**Neural-like P System**)**.** *Formally, a* neural-like P system*, of degree* $m \geq 1$*, is a construct*

$$\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, syn, i_0),$$ *where*

1. O *is a finite non-empty alphabet (of* objects*, usually called* impulses*;*

2. $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ *are* cells *(also called* neurons*), of the form*

$$\sigma_i = (Q_i, S_{i,0}, w_{i,0}, R_i), 1 \leq i \leq m,$$

where

(a) Q_i *is a finite set (of* states*);*

(b) $S_{i,0} \in Q_i$ *is the* initial state*;*

(c) $w_{i,0} \in O^*$ *is the* initial multiset *of impulses of the cell;*

(d) R_i *is a finite set of rules of the form* $sw \to s'xy_{go}z_{out}$*, where* $s, s' \in Q_i$*,* $w, x \in O^*$*,* $y_{go} \in (O \times \{go\})^*$ *and* $z_{out} \in (O \times \{out\})^*$ *with the restriction that* $z_{out} = \lambda$ *for all* $i \in \{1, 2, \ldots, m\}$ *different from* i_0*;*

3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ *(*synapses *among cells);*

4. $i_0 \in \{1, 2, \ldots, m\}$ *indicates the* output cell.

The standard rules used in this model are of the form $sw \to s'w'$, where s, s' are states and w, w' are multisets of impulses. The mark "go" assigned to some elements of w' means that these impulses have to leave immediately the cell and pass to the cells to which we have direct links through synapses. The communication among the cells of the system can be done in a replicative manner (the same object is sent to all adjacent cells), or in a non-replicative manner (the impulses are sent to only one neighbouring cell, or can be distributed non-deterministically to the cells to which we have synapses). The objects marked with "out" (they can appear only in the cell i_0) leave the system. The computation is successful only if it halts, reaches a configuration where no rule can be applied.

The sequence of objects (note that they are symbols from an alphabet) sent to the environment from the output cell is the string computed by a halting computation, hence the set of all strings of this type is the language computed/generated by the system.

Several ingredients of a neural-like P system are modified in spiking neural P system, bringing the model closer to the way the neurons communicate by means of spikes.

2.6.4 Spiking Neural P Systems (SN P Systems)

Spiking neural P systems [50] (shortly called SN P systems) are parallel and distributed computing models inspired by the neurophysiological behaviour of neurons sending electrical pulses of identical voltages called spikes to the neighbouring neurons through synapses. It is represented as a directed graph where nodes correspond to the neurons having spiking and forgetting rules that involve the spikes present in the neuron in the form of occurrences of a symbol a. The arcs indicate the synapses among the neurons. Formally, a spiking neural P system is defined as follows.

Definition 2.7 (**An SN P system**)**.** *Mathematically, we represent a spiking neural P system, of degree* $m \geq 1$*, in the form*

$$\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, syn, i_0)\text{, where}$$

1. $O = \{a\}$ *is the singleton alphabet (*a *is called* spike*) ;*

2. $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ *are neurons, of the form*

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

where

(a) $n_i \geq 0$ *is the* initial number of spikes *contained by the cell;*

(b) R_i *is a finite set of* rules *of the following two forms:*

i. $E/a^r \rightarrow a; d$*, where* E *is a regular expression over* O*,* $r \geq 1$*, and* $d \geq 0$*; Number of spikes present in the neuron is described by the regular expression* E*,* r *spikes are consumed and it produces a spike, which will be sent to other neurons after* d *time units*

ii. $a^s \rightarrow \lambda$*, for some* $s \geq 1$*, with the restriction that* $a^s \notin L(E)$ *for any rule* $E/a^r \rightarrow a; d$ *of type (i) from* R_i*;*

3. $syn \subseteq \{1, 2, 3, \ldots, m\} \times \{1, 2, 3, \ldots, m\}$ ***with*** $(i, i) \notin syn$ ***for*** $1 \leq i \leq m$ *(*synapses *among cells);*

4. $i_0 \in \{1, 2, 3, \ldots, m\}$ *indicates the output neuron.*

The rules of type $E/a^r \rightarrow a; d$ are spiking rules, and can be applied only if the neuron contains n spikes such that $a^n \in L(E)$ and $n \geq r$. When neuron σ_i spikes, its spike is replicated in such a way that one spike is sent to all neurons σ_k such that $(i, k) \in syn$, and σ_k is open at that moment. If $d = 0$, then the spikes are emitted immediately, if $d = 1$, then the spikes are emitted in the next step and so on. In the case $d \geq 1$, if the rule is used in step p, then in step $p, p + 1, p + 2, \ldots, p + d - 1$, the neuron is closed and it cannot receive new spikes (If a neuron has a synapse to a closed neuron and sends spikes along it, then the spikes are lost; that reflects the refractory period of biological neurons). In step $p + d$, the neuron spikes and becomes open again, hence can receive spikes (which can be used in step $p + d$). If a neuron σ_i fires and either it has no outgoing synapse, or all neurons σ_k such that $(i, k) \in syn$ are closed, then the spike of neuron σ_i is lost; the firing is allowed, it takes place, but results in no new spikes.

If $d = 0$, then sometimes it is omitted when writing the rule and are called non delayed rules. We also consider SN P systems without delays (i.e. $d = 0$ in all rules).

The rules of type $a^s \rightarrow \lambda$ are forgetting rules; s spikes are simply removed ("forgotten") when applying. Like in the case of spiking rules, the left hand side of a forgetting rule must "cover" the contents of the neuron, that is, $a^s \rightarrow \lambda$ is applied only if the neuron contains exactly s spikes. 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. In each neuron σ_i with k rules, the rules are numbered as ***i1***, ***i2***, . . ., ***ik***.

If we have a rule $E/a^r \rightarrow a; d$ with $L(E) = a^r$, then we write simply $a^r \rightarrow a; d$. If all rules are of this form, then the system is called *bounded* (or *finite*), because it can

handle only finite numbers of spikes in the neurons.

The *configuration* of the system is described by both the contents of each neuron and its state, which can be expressed as the number of steps to wait until it becomes open (zero if the neuron is already open). Thus $\langle \alpha_1/d_1, \alpha_2/d_2, \ldots, \alpha_m/d_m \rangle$ is a configuration where neuron σ_i contains $\alpha_i \geq 0$ spikes and it will open after $d_i \geq 0$ steps, for $i = 1, 2, 3, \ldots, m$. With this notation, the initial configuration of the system is described by C_0=$\langle n_1/0, n_2/0, n_3/0, \ldots, n_m/0 \rangle$.

The SN P system is synchronized by means of a global clock and works in a locally sequential and globally maximal manner. That is, the working is sequential at the level of each neuron. In each neuron, at each step, if there are more than one rule is enabled by its current contents, then only one of them (chosen nondeterministically) can fire. But still, the system as a whole evolves in parallel and in a synchronising way, as in, at each step, all the neurons (that have an enabled rule) choose a rule and all of them fire at once.

Using the rules, we pass from one configuration of the system to another configuration, such a step is called a *transition*. A *computation* of Π is a finite or infinite sequence of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called reachable.

With any computation halting or not we associate a spike train, a sequence of digits of 0 and 1, with 1 appearing in position which indicates the steps when the output neuron sends spikes out of the system. One of the neurons is considered to be the output neuron, and its spikes are sent to the environment.

In the initial paper on SN P systems [50], the result was often defined as the distance between the first two spikes of a spike train. We denote by $N_2(\Pi)$ the set of numbers computed by considering the distance between the first two spikes of a spike train, with the subscript 2 reminding of the way the result of a computation is defined, and by $N_2SNP_m(rule_k, cons_p, forg_q)$ the family of all sets $N_2(\Pi)$ computed

as above by spiking neural P systems with at most $m \geq 1$ neurons, using at most $k \geq 1$ rules in each neuron, with all spiking rules $E/a^r \to a; d$ having $r \leq p$, and all forgetting rules $a^s \to \lambda$ having $s \leq q$. When one of the parameters m, k, p, q is not bounded, then it is replaced with $*$.

Then in [19] several extensions were examined. The distance between the first k spikes of a spike train, or the distances between all consecutive spikes, taking into account all intervals or only intervals that alternate, all computations or only halting computations, etc. The way one generally defines the result in membrane computing can also be considered in SN P systems, namely, the result is taken to be the total number of spikes collected in the output neuron (or the environment) during a computation at moment the computation halts.

Moreover, the result can obviously be defined as the spike train itself. In this way, SN P systems are used as binary language generators. The SN P systems mentioned above all work in the generating mode: starting from a fixed initial configuration the nondeterministic features of the system make it run in different computations, and the output of these computations is collected.

Actually, an SN P system can also work in the accepting mode: no output neuron is needed, but instead an input neuron is designated. Then as an encoding of input n, two consecutive spikes are introduced in this input neuron, at an interval of n time steps; the number n is accepted if the resulting computation halts.

When both input and output neurons are considered, the SN P system can be used as a transducer, both for strings and infinite sequences, as well as for computing numerical functions. Spikes can be introduced in the input neuron, at various steps, while the spikes of the output neuron are sent to the environment. A binary sequence is associated with the spikes entering or exiting the system. In the transducer mode, a large class of (Boolean) functions can be computed. They are also used as computation devices that solve computationally difficult (**NP**-complete) decision problems [31].

SN P Systems as Language Generators

In the generative mode, the system has no input neurons and one of the neurons is considered to be the output neuron, and its spikes are sent to the environment.

Let Π=(O, σ_1, σ_2, $\sigma_3, \ldots, \sigma_m$, syn , i_0), be an SN P system and let $\gamma = \mathcal{C}_0 \Longrightarrow \mathcal{C}_1 \Longrightarrow \ldots \Longrightarrow \mathcal{C}_k$ be an halting computation. ($\mathcal{C}_0$ is the initial configuration, and $\mathcal{C}_{i-1} \Longrightarrow \mathcal{C}_i$ is the ith step of γ). Let us denote by $bin(\gamma)$ the string $b_1 b_2 \ldots b_k$ where $b_i \in \{0, 1\}$ and $b_i = 1$ if and only if the (output neuron of the) system Π sends a spike into the environment in step i of γ otherwise $b_i = 0$. We denote by B the binary alphabet $\{0, 1\}$, and by $COM(\Pi)$ the set of all halting computations of Π. Moreover, we define the language generated by Π by $L(\Pi) = \{bin(\gamma) \mid \gamma \in COM(\Pi)\}$.

The complexity of an SN P system Π is described as $LSNP_m(rule_k, cons_p, forg_q)$, the family of languages $L(\Pi)$, generated by systems Π with at most m neurons, each neuron having at most k rules, each of the spiking rules consuming at most p spikes and each forgetting rule removing at most q spikes. As usual a parameter m, k, p, q is replaced with $*$ if it is not bounded. If the SN P systems are finite (i.e., contain only a bounded number of spikes), we denote the corresponding families of languages by $LFSNP_m(rule_k, cons_p, forg_q)$.

Let us mention some results about languages generated by SN P systems [18].

Theorem 2.5. *(i) There are finite languages (for instance, $\{0^k, 10^j\}$, for any $(k \geq 1, j \geq 0)$, which cannot be generated by any SN P system, but for any $L \in FIN, L \subseteq B^+$, we have $L\{1\} \in LFSNP_1(rule_*, cons_*, forg_0)$ and if $L \in FIN, L \subseteq B^+, L = \{x_1, x_2, \ldots, x_n\}$, then $\{0^{i+3}x_i \mid 1 \leq i \leq n\} \in LFSNP_*(rule_*, cons_1, forg_0)$.*
(ii) The family of languages generated by finite SN P systems is strictly included in the family of regular languages over the binary alphabet, but for any regular language $L \subseteq V^*$ there is a finite SN P system Π and a morphism $h : V^* \to B^*$ such that $L = h^{-1}(L(\Pi))$.
(iii) $LSNP_*(rule_*, cons_*, forg_*) \subset REC$, but for every alphabet $V = \{a_1, a_2, \ldots, a_k\}$

there are two symbols b, c *not in* V*, a morphism* $h_1 : (V \cup \{b, c\})^* \to B^*$ *and a projection* $h_2 : (V \cup \{b, c\})^* \to V^*$ *such that for each language* $L \subseteq V^*, L \in RE$*, there is an SN P system* Π *such that* $L = h_2({h_1}^{-1}(L(\Pi)))$.

These results show that the language generating power of SN P systems is rather eccentric; on the one hand, finite languages (like $\{0, 1\}$) cannot be generated, on the other hand, we can represent any RE language as the direct morphic image of an inverse morphic image of a language generated in this way. This eccentricity is due mainly to the restricted way of generating strings, with one symbol added in each computation step.

Example 2.2.

We will illustrate some definitions of standard SN P system with an example from Section 5 in [50]. Figure 2.3(a) represents the initial configuration of the SN P system Π_1. We have three neurons, labelled with $1, 2, 3$, with neuron 3 being the output one. The neurons are represented by nodes of a directed graph whose arrows represent the synapses; an arrow also exits from the output neuron, pointing to the environment; in each neuron we specify the rules and the spikes present in the initial configuration. Neuron σ_1 is having one spiking rule and one forgetting rule. The rule $a^2/a \to a$ fires if it contains two spikes; one spike is consumed, the other remains available for the next step. Neuron σ_2 is having two same rules (firing rules which can be chosen in a non-deterministic way, the difference between them being in the delay from firing to spiking), and neuron σ_3 having two firing and one forgetting rule. It is formally represented as:

Π_1 = ($\{a\}$, σ_1, σ_2, σ_3, syn, 3), with
σ_1 = (2, $\{a^2/a \to a; 0, a \to \lambda\}$),
σ_2 = (1, $\{a \to a; 0, a \to a; 1\}$),
σ_3 = (3, $\{a^3 \to a; 0, a \to a; 1, a^2 \to \lambda\}$),
syn = {(1,2),(2,1),(1,3),(2,3)}.

The initial configuration of the system is $< 2/0, 1/0, 3/0 >$. It works as follows. All neurons can fire in the first step, with neuron σ_2 choosing non- deterministically between its two rules.

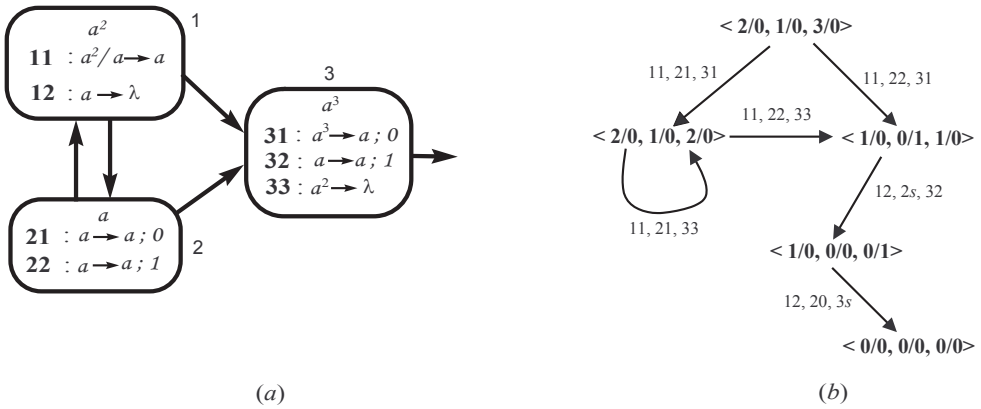


Figure 2.3: (a) An SN P system Π_1 (b) Evolution of Π_1

Output neuron σ_3 sends its spike to the environment. If the neuron σ_2 uses its first rule, then both the neurons σ_1 and σ_2 exchange their spikes and send a spike to the output neuron σ_3 and we reach the configuration $< 2/0, 1/0, 2/0 >$. As long as neuron σ_2 uses the rules $a \rightarrow a; 0$, the computation cycles in the same configuration: neurons σ_1 and σ_2 exchange spikes, while σ_3 forgets its two spikes.

However, at any moment, starting with the first step of the computation, σ_2 can choose to use the rule $a \rightarrow a; 1$. This means that the spike of σ_1 cannot enter σ_2, it only goes to σ_3; in this way, σ_2 will never work again because it remains empty and we reach the configuration $< 1/0, 0/1, 1/0 >$. Here neuron σ_1 has to use its forgetting rule $a \rightarrow \lambda$, while neuron σ_3 fires, using the rule $a \rightarrow a; 1$. Simultaneously, σ_2 emits its spike, but it cannot enter σ_3 (it is closed this moment); the spike enters neuron σ_1 and we get the configuration $< 1/0, 0/0, 0/1 >$. The spike in σ_1 is forgotten in the next step. In this way, no spike remains in the system and we reach the halting configuration $< 0/0, 0/0, 0/0 >$. The computation ends with the expelling of the

spike from neuron σ_3. Because of the waiting moment imposed by the rule $a \to a; 1$ from σ_3, the two spikes of this neuron cannot be consecutive, but at least two steps must exist in between. Thus, we conclude that Π computes all natural numbers greater than or equal to two. Thus, we conclude that

$$N_2(\Pi_1) = \mathbb{N}^+ - \{1\} \in N_2SNP_3(rule_3, cons_3, forg_2)$$

The evolution of the system Π_1 can be analyzed on a transition diagram as that from Figure 2.3(b): because the system is finite, the number of configurations reachable from the initial configuration is finite, too, hence, we can place them in the nodes of a graph, and between two nodes/configurations we draw an arrow if and only if a direct transition is possible between them. In Figure 2.3(b) we have also indicated the rules used in each neuron, with the following conventions: **ij** denotes the jth rule in neuron σ_i, with 31 being written in italics, in order to indicate that a spike is sent out of the system at that step; when a neuron $i = 1, 2, 3$ uses no rule, we have written $i0$, and when it spikes (after being closed for one step), we write is.

The transition diagram of a finite SN P system can be interpreted as the representation of a non-deterministic finite automaton, with C_0 being the initial state, the halting configurations being final states, and each arrow being marked with 0 if in that transition the output neuron does not send a spike out, and with 1 if in the respective transition the output neuron spikes; in this way, we can identify the language generated by the system. In the case of the finite SN P system Π_1, the generated language is $L(\Pi_1) = L(1(0^+ + \lambda)01)$.

Normal Forms and Universality Results

In the initial definition of SN P systems several ingredients are used (delay, forgetting rules), some of them of a general form (general synapse graph, general regular expressions). As shown in [46, 80], rather restrictive normal forms can be found,

in the sense that some ingredients can be removed or simplified without losing the computational completeness. For instance, the forgetting rules or the delay can be removed, both the indegree and the outdegree of the synapse graph can be bounded by 2, while the regular expressions from firing rules can be of very restricted forms.

The following universality results were proved in [50] and extended in [34] to other ways of defining the result of a computation.

Theorem 2.6. *1.* $NFIN = N_2SNP_1(rule_*, cons_1, forg_0) = N_2SNP_2(rule_*, cons_*, forg_*)$.

2. $NRE = N_2SNP_*(rule_k, cons_p, forg_q)$, ***for all*** $k \geq 2, p \geq 3, q \geq 3$.

3. $NSLIN = N_2SNP_*(rule_k, cons_p, forg_q, bound_s)$, ***for all*** $k \geq 3, q \geq 3, p \geq 3, s \geq 3$.

There is a universal computing SN P system with standard rules without delay having 11 neurons [78], and one with extended rules which has 3 neurons [77].

SN P Systems as Transducers

Standard spiking neural P systems in transducer mode can simulate the Boolean circuits [52], with two spikes sent out of the system encoded as 1 and one spike as 0. Boolean circuits are constructed using these fundamental gates and synchronising SN P system to establish synchronization among the gates to output the correct result.

In [66], a uniform solution to the SAT (in CNF, with n variables and m clauses) is provided using standard SN P systems without delay having $3n^2 + 8m + 5$ neurons, providing the solution in a number of steps which is linear in the number of variables. Two bits were used to code each literal of a clause, hence the computation cannot end in less than $2n$ steps.

2.6.5 SN P Systems with Anti-Spikes (SN PA Systems)

We discuss SN PA systems [79] in detail as these variants were considered in the thesis for translation.

Definition 2.1 *(SN P systems with anti-spikes)* A spiking neural P system with anti-spikes, of degree $m \geq 1$, is a construct

$$\Pi=(O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, \textit{syn}, \textit{in}, \textit{out}), \text{ where}$$

1. $O = \{ a, \overline{a} \}$ is the binary alphabet. a is called *spike* and $\overline{a}$ is called *anti-spike*.
2. $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ are neurons, of the form

$$\sigma_i=(n_i, R_i), 1 \leq i \leq m, \text{ where}$$

   (a) $|n_i|$ is the number of spikes or anti-spikes contained in the neuron σ_i and if $n_i > 0$ then the neuron is having n_i spikes and if $n_i < 0$ then the neuron is having n_i anti-spikes;

   (b) R_i is a finite set of *rules* of the following two forms:

      i. $E/b^r \rightarrow b'$, where E is a regular expression over a or $\overline{a}$, while $b, b' \in \{a, \overline{a}\}$, and $r \geq 1$.

      ii. $b^s \rightarrow \lambda$, where λ is the empty word, $s \geq 1$, $b \in \{a, \overline{a}\}$ and for all $E/b^r \rightarrow b'$ from R_i, $b^s \notin L(E)$ where $L(E)$ is the language defined by E.

3. $\textit{syn} \subseteq \{ 1, 2, 3, \ldots, m\} \times \{ 1, 2, 3, \ldots, m\}$ with $(i, i) \notin \textit{syn}$ for $1 \leq i \leq m$ (*synapses* among cells);
4. $\textit{in}, \textit{out} \in \{1, 2, 3, \ldots, m\}$ are the input and output neurons respectively.

The rules of type $E/b^r \rightarrow b'$ are spiking rules, and they are used only if the neuron contains n bs such that $b^n \in L(E)$ and $n \geq r$. When neuron σ_i sends b' (a spike/anti-spike), it is replicated in such a way that one spike/anti-spike is sent to all neurons σ_k such that $(i, k) \in syn$.

The rules of type $b^s \to \lambda$ are the forgetting rules; s spikes/anti-spikes are simply removed ("forgotten") when applying the rule. Like in the case of spiking rules, the left hand side of a forgetting rule must "cover" the contents of the neuron, that is, $a^s \to \lambda$ is applied only if the neuron contains exactly s spikes.

A spike/anti-spike emitted by neuron σ_i will pass immediately to all neurons σ_k such that $(i,k) \in syn$. That means transmission of spikes/anti-spikes takes no time (since the rules are non delayed rules), the spikes/anti-spikes will be available in neuron σ_k in the next step. There is an additional fact that a and $\overline{a}$ cannot stay together, they annihilate each other. If a neuron has either objects a or objects $\overline{a}$, and further objects of either type (maybe both) arrive from other neurons, such that we end with a^r and $\overline{a}^s$ inside, then immediately an annihilation rule $a\overline{a} \to \lambda$ (which is implicit in each neuron), is applied in a maximal manner, so that either a^{r-s} or $(\overline{a})^{s-r}$ remain for the next step, provided that $r \geq s$ or $s \geq r$, respectively. This mutual annihilation of spikes and anti-spikes takes no time and the annihilation rule has priority over spiking and forgetting rules, so each neuron always contains either only spikes or anti-spikes. Like in [79], we avoid using rules $\overline{a}^c \to \overline{a}$, but not the other three types, corresponding to the pairs (a,a), $(a,\overline{a})$, $(\overline{a},a)$. If we have a rule $E/b^r \to b'$ with $L(E) = \{b^r\}$, then we write it in the simplified form as $b^r \to b'$.

The *configuration* of the system is described by $\mathcal{C} = \langle \beta_1, \beta_2, \ldots, \beta_m \rangle$, where β_i is a the number of spikes/anti-spikes present in neuron σ_i. The initial configuration is $\mathcal{C}_0 = \langle n_1, n_2, \ldots, n_m \rangle$.

A global clock is assumed and in each time unit, each neuron which can use a rule should do it (the system is synchronized), but the work of the system is sequential locally: only (at most) one rule is used in each neuron except the annihilation rule which fires maximally with highest priority. For example, if a neuron σ_i has two firing rules, $E_1/a^r \to a$ and $E_2/a^k \to a$ with $L(E_1) \cap L(E_2) \neq \emptyset$, then it is possible that each of the two rules can be applied, and in that case only one of them is chosen non-deterministically. Thus, the rules are used in the sequential manner in each

neuron, but neurons function in parallel with each other. In each step, all neurons which can use a rule of any type, spiking or forgetting, have to evolve, using a rule.

Using the rules in this way, we pass from one configuration of the system to another configuration; such a step is called a transition. For two configurations $\mathcal{C}$ and $\mathcal{C}'$ of Π we denote by $\mathcal{C} \Longrightarrow \mathcal{C}'$, if there is a direct transition from $\mathcal{C}$ to $\mathcal{C}'$ in Π.

A computation of Π is a finite or infinite sequence of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called reachable. A computation halts if it reaches a configuration where no rule can be used. An SN PA system can be used as a computing device in various ways. In the thesis, we use them as language generators and transducers. In the generative mode, one of the neuron is considered as output neuron and it sends output to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1, the moments of time when an anti-spike emitted is marked with 0 and no output moments are just ignored. This binary sequence is called the spike train of the system- it might be infinite if the computation does not stop. With halting configurations, we associate a language, the binary strings describing the spike trains.

Let $\gamma = \mathcal{C}_0 \Longrightarrow \mathcal{C}_1 \Longrightarrow \ldots \Longrightarrow \mathcal{C}_k$ be an halting computation. Let us denote by $bin(\gamma)$ the string $b_1 b_2 \ldots b_k$ where $b_i \in \{0, 1\}$ and $b_i = 1$ iff the output neuron of the system Π sends a spike into the environment in the step i of γ, $b_i = 0$ iff it sends an anti-spike, and $b_i = \lambda$ if the step i generated no output. We denote by B the binary alphabet $\{0, 1\}$ and by $COM(\Pi)$, the set of all halting computations of Π. Moreover, we define the language generated by Π by $L(\Pi) = \{bin(\gamma) \mid \gamma \in COM(\Pi)\}$.

The complexity of an SN PA system Π is described as $LSNPA_m$ ($rule_k$,$cons_{p_1,p_2}$, $forg_{q_1,q_2}$), the family of languages $L(\Pi)$, generated by systems Π with at most m neurons, each neuron having at most k rules, each of the spiking rules consuming at most p_1 spikes and p_2 anti-spikes and each forgetting rule removing at most q_1 spikes and q_2 anti-spikes. As usual a parameter m, k, p_1, p_2, q_1, q_2 is replaced with $*$ if it is not

bounded. If the underlying SN PA systems are finite, we denote the corresponding families of languages by $LFSNPA_m(rule_k, cons_{p_1,p_2}, forg_{q_1,q_2})$.

Example 2.3.

Consider the graphical representation of an SN P system with anti-spikes Π_4 in Figure 2.4(a). It is formally denoted as

$\Pi_4 = (O, \sigma_1, \sigma_2, syn, 2)$, with

$\sigma_1 = (-1, \{\overline{a} \to a\})$,

$\sigma_2 = (2, \{a^2/a \to \overline{a}, a^2 \to a\})$,

$syn = \{(1,2), (2,1)\}$.

The evolution of the system Π_4 can be analysed on a transition diagram as that from

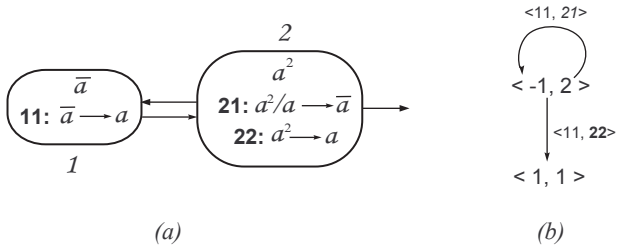


Figure 2.4: SN P system with anti-spikes Π_4 generating 0^*1

Figure 2.4(b) because the system is finite, the number of configurations reachable from the initial configuration is finite too, hence, we can place them in the nodes of a graph and between two nodes/configurations we draw an arrow if and only if a direct transition is possible between them. In the Figure 2.4(b), we have also indicated the rules used in each neuron with the following conventions; for each ***ij*** we have written only the subscript ij with *21* written in italics and **22** in bold in order to indicate that an anti-spike is sent to environment at steps when *21* is used and a spike when **22** is used; $i0$ is written when a neuron σ_i uses no rule.

The functioning can easily be followed on this diagram, so that we only briefly describe it. We have two neurons, with labels 1, 2; neuron σ_2 is the output neuron.

Initially neuron σ_1 has one anti-spike with a rule and σ_2 has two spikes with two rules and non-determinism between its two rules. So the initial configuration of the system, $\mathcal{C}_0 =< -1, 2 >$.

The two neurons fire in the first step. Neuron σ_1 uses its rule $\overline{a} \rightarrow a$ and sends a spike (1) to σ_2. Neuron σ_2 can choose any of its two rules and as long as it uses first rule, one spike is consumed and an anti-spike is sent to σ_1 and the environment. In the next step the system will be in the same configuration. At any instance of time, starting from step 1, σ_2 can choose its second rule, which consumes its two spikes and sends a spike to σ_1 and environment. In the next step each neuron will have one spike, reaching the configuration $< 1, 1 >$ and the systems halts.

Similar to the SN P system, the transition diagram of a finite SN PA system can also be interpreted as the representation of a non-deterministic finite automaton, with $\mathcal{C}_0$ being the initial state, the halting configurations being final states and each arrow being marked with 0 if in that transition the output neuron sends an anti-spike and with 1 if it sends a spike. In this way, we can identify the language generated by the system. So the language generated by the SN PA system Π_4 is 0^*1.

2.6.6 Other Variants of SN P Systems

The derivation mode considered in the initial paper of the SN P systems is the maximal strategy (locally sequential and globally maximal strategy). Several SN P systems are introduced by considering other derivation modes. Different variants of SN P systems are also developed by researchers by adding features inspired by nature such as axons, neuron division, astrocytes, and inhibitory impulses. Apart from the SN P systems described below, there were investigated several other types of SN P systems: with several output neurons [47] and with packages of spikes sent along specified synapse links ([9]) etc. We refer the reader to P systems web page at [2].

Extended SN P Systems

An extended spiking neural P system [8] has more general rules of the form $E/a^r \rightarrow a^p; d$, where $r \geq 1$ and $r \geq p \geq 0$. Such rules operate in the same manner as spiking rules of standard SN P systems except that firing sends p spikes along the each outgoing synapses (and after d time steps, these p spikes are received simultaneously by each neighbouring neuron). Note when $p = 1$, the rule $E/a^r \rightarrow a^p; d$ is reduced to standard spiking rule and when $p = 0$, the rule becomes a forgetting rule.

Languages - even on arbitrary alphabets - can be obtained using extended rules. In this case, a language can be generated by associating the symbol b_i with a step when the output neuron sends out i spikes, with an important decision to take in the case $i = 0$: we can either consider b_0 as a separate symbol, or we can assume that emitting 0 spikes means inserting λ in the generated string. Thus, we both obtain strings over arbitrary alphabets, not only over the binary one, and, in the case where we ignore the steps when no spike is emitted, a considerable freedom is obtained in the way the computation proceeds. This latter variant (with λ associated with steps when no spike exits the system) is considered below.

We denote by $LSN^eP_m(rule_k, cons_p, prod_q)$ the family of languages $L(\Pi)$, generated by SN P systems Π using extended rules, with the parameters m, k, p, and q similar as standard SN P systems. The next counterparts of the results from Theorem 2.6 were proved in [19].

Theorem 2.7. *(i)* $FIN = LSN^eP_1(rule_*, cons_*, prod_*)$ ***and this result is sharp, as*** $LSN^eP_2(rule_2, cons_2, prod_2)$ ***contains infinite languages.***
(ii) $LSN^eP_1(rule_*, cons_*, prod_*) \subseteq REG \subset LSN^eP_3(rule_*, cons_*, prod_*)$***; the second inclusion is proper, because*** $LSN^eP_3(rule_3, cons_4, prod_2) - REG \neq \emptyset$***; actually,*** $LSN^eP_3(rule_3, cons_6, prod_4)$ ***contains non-semilinear languages.***
(iii) $RE = LSN^eP_*(rule_*, cons_*, prod_*)$.

Asynchronous SN P Systems

The standard SN P system model relies on the fact that all neurons fire in each step where they are fireable. This synchronization is quite powerful so it is of interest to study the power of SN P systems with lesser requirements. An asynchronous SN P system (introduced in [17]) is an SN P system model which does not require the neurons to fire in a given time frame. During each step, any number of fireable neurons are fired (including the possibility of firing no neurons). When a neuron is fireable it may (or may not) choose to fire during the current step. If the neuron chooses not to fire, it may fire in any later step as long as the rule is still applicable. (The neuron may still receive spikes while it is waiting which may cause the neuron to no longer be fireable.) Hence there is no restriction on the time interval for firing a neuron. Once a neuron chooses to fire, the appropriate number of spikes are sent out after a delay of exactly d time steps and are received by the neighbouring neurons during the step when they are sent.

Sequential SN P Systems

Sequential SN P systems [49] require one and only one neuron to fire per step when the system is not dormant. When the system is dormant, no neuron fires in the current step (but the computation progresses by decreasing the remaining delay for each closed neuron by one). We will investigate the computational power of sequential SN P systems whose behaviour is controlled to operate in a sequential manner. More precisely, the SN P system is restricted in its operation as follows:

- As before, the system starts from a fixed initial configuration and is synchronized, i.e., there is a global clock (so all the neurons use this clock).
- However, a step consists of nondeterministically choosing a "fireable" rule. (For convenience, a forgetting rule is classified as a "fireable" rule.) If there

is no fireable rule, then the system is dormant until a rule becomes fireable. However, the clock will keep on ticking.

- The convention for halting is like before, i.e., all neurons are open and none are fireable.

SN P Systems with Exhaustive use of Rules

In the usual SN P system (either standard SN P system or extended as discussed above), although the system works in parallel at the level of all neurons (i.e., the system works in the synchronous manner), at the level of each neuron only (at most) one rule can be applied. In the SN P systems working in the exhaustive manner [51], when a rule is applied, then it must be applied as many times as possible in that neuron so that the neuron remains as few spikes as possible. Under the exhaustive mode, not only the system works in parallel at the level of all neurons, but also a rule is applied in a maximally parallel manner at the level of each separate neuron.

SN P systems with Astrocytes

An important ingredient of neurobiology is missing from SN P systems: the astrocytes. They are cells which play an essential role in the functioning and interaction of neurons, by feeding them differently with nutrients depending on their individual activity. More specifically, astrocytes are cells which sense at the same time the spike traffic along several neighbouring axons, and feed the respective neurons (e.g., with calcium) depending on the spikes frequency. The first attempt to introduce this kind of cells into SN P systems was made by Binder et al. in [12]. In this model, astrocytes have the role of "excitatory" and "inhibitory" according to the quantity of spikes passing through the synapse in one time. Also, a simple model is considered by Păun in [32], where astrocytes only have the role of "inhibitory". In such a model, the system works in a rather restricted manner: An astrocytes checking several

axons leaves to pass only one spike along them, suppressing all others. Obviously. SN P systems with astrocytes have stronger computing power than the general SN P systems, thus they are also computationally complete. However they add a new powerful feature so that this kind of models has a significant potential to be applied to various problems in terms ease of programming and in terms of computational complexity.

SN P Systems with Neuron Division and Budding

The biological motivation for the mechanism of neuron division and budding that we introduce into SN P systems comes from the recent discoveries in neurobiology related to neural stem cells. Neural stem cells are persistent throughout life within central nervous system in the adult mammalian brain, which ensures a life-long contribution of new neurons to a self-renewing nervous system with about 30000 new neurons being produced every day. These observations are incorporated in SN P systems by considering neuron division and budding, and by providing a "synapse dictionary" according to which new synapses are generated. In [81], SN P systems with neuron division and budding solved computationally hard problems, where for all $n, m \in \mathbb{N}^+$ all the instances of SAT(n, m) with at most n variables and at most m clauses are solved in a deterministic way in polynomial time using a polynomial number of initial neurons.

2.7 Petri Nets

Petri nets are graphical and mathematical modelling tools applicable to systems that are characterized as being concurrent, distributed, non-deterministic and parallel. As a mathematical tool, it provides a set of state equations, algebraic equations and other mathematical models governing the behaviour of the systems. A Petri net

is graphically represented as a directed weighted bipartite graph consisting of two kinds of nodes called *places* and *transitions*, where *arcs* are either from places to transitions or from transitions to places. The places are drawn as circles, transitions as bars or boxes. Arcs are labelled with their weights, where a k-weighted arc can be interpreted as the set of k parallel arcs. Labels for unit weight arcs are usually omitted. In order to study the dynamic behaviour of a Petri net modelled system in terms of its states and state changes, each place may potentially hold either none or a positive number of *tokens*.

2.7.1 Formal Definition

The formal definition of the Petri net that we start with is essentially the same as that in [76].

Definition 2.8 (**P/T net**). *A P/T net (also known as a place/transition net or a Petri net) is a 5-tuple* $\mathcal{N} = (P, T, F, W, \mathcal{M}_0)$*, where*
$P = \{p_1, p_2, \ldots p_m\}$ *is a finite, non-empty set of places;*
$T = \{t_1, t_2, \ldots t_n\}$ *is a finite, non-empty set of transitions;*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs;*
$W : F \to \mathbb{N}^+$ *is a weight function; and*
$\mathcal{M}_0 : P \to \mathbb{N}$ *is the initial marking.*

A place p is an *input* (or an *output*) place of a transition t iff there exists an arc (p, t) (or (t, p), respectively) in the set F. The sets of all input and output places of a transition t are denoted by $I(t) = \{p \mid (p, t) \in F\}$ and $O(t) = \{p \mid (t, p) \in F\}$, respectively. Similarly the set of all input and output transitions of a place p is defined as $I(p) = \{t \mid (t, p) \in F\}$ and $O(p) = \{t \mid (p, t) \in F\}$.

A transition without any input place is called a *source transition*, and one without any output place is called a *sink transition*. Note that a source transition is unconditionally enabled, and that the firing of a sink transition consumes tokens,

but doesn't produce any. A pair of a place p and a transition t is called a *self-loop*, if p is both an input place and an output place of t. A Petri net is said to be *pure* if it has no self-loops.

Definition 2.9 (**Marking**). *A marking of a net $\mathcal{N} = (P, T, F, W, \mathcal{M}_0)$ is a function $\mathcal{M} : P \to \mathbb{N}$, i.e. a multiset over P. Graphically, a marking is expressed using a respective number of black tokens in each place. Submarking of a Petri net is the marking of some of its places.*

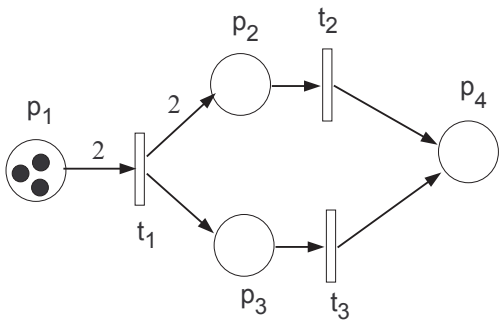
A state or marking in a Petri net is changed according to the *occurrence (firing) rules*:

Definition 2.10 (**Occurrence rule**). *Let $\mathcal{N} = (P, T, F, W, \mathcal{M}_0)$ be a P/T net. A transition $t \in T$ is enabled to occur in a marking $\mathcal{M}$ of $\mathcal{N}$ iff $\mathcal{M}(p) \geq W(p,t)$ for every place $p \in I(t)$. If a transition t is enabled to occur in a marking $\mathcal{M}$, then its occurrence leads to the new marking $\mathcal{M}'$ defined by*
$\mathcal{M}'(p) = \mathcal{M}(p) - W(p,t) + W(t,p)$ for every $p \in P$. We write $\mathcal{M}[t\rangle$ to denote that t may fire in $\mathcal{M}$, and $\mathcal{M}[t\rangle\mathcal{M}'$ to indicate that the firing of t in $\mathcal{M}$ leads to $\mathcal{M}'$. In the same way, we write $\mathcal{M}\overline{[t\rangle}$ to denote that t cannot fire in $\mathcal{M}$.

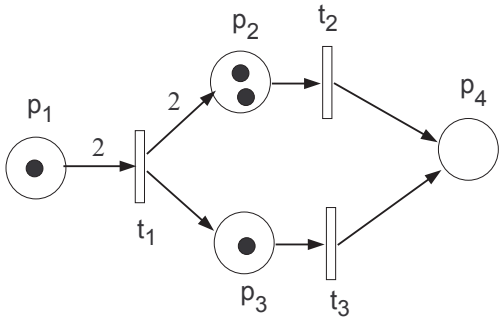
Example 2.4.

Consider the Petri net $\mathcal{N}_1 = (P, T, F, W, \mathcal{M}_0)$, where each component is given as
$P = \{p_1, p_2, p_3, p_4\}$;
$T = \{t_1, t_2, t_3\}$;
$F = \{(p_1,t_1), (t_1,p_2), (t_1,p_3), (p_2,t_2), (p_3,t_3), (t_2,p_4), (t_3,p_4)\}$;
$W(p_1,t_1) = W(t_1,p_2) = 2$,
$W(t_1,p_3) = W(p_2,t_2) = W(p_3,t_3) = W(t_2,p_4) = W(t_3,p_4) = 1$;
$\mathcal{M}_0 = (3,0,0,0)$ is the initial marking.

Figure 2.5(a) shows the Petri net graph for $\mathcal{N}_1$.
Under the initial marking, $\mathcal{M}_0 = (3,0,0,0)$, only t_1 is enabled. Firing of t_1 results in a



(*a*) A Petri net with initial marking



(b) Petri net after firing of transtion t_1

Figure 2.5: A simple Petri net and an illustration of transition firing.

new marking, say $\mathcal{M}_1$. It follows from the firing that $\mathcal{M}_1 = (1, 2, 1, 0)$. The new token distribution of this Petri net is shown in Figure 2.5(b).
Again, in marking $\mathcal{M}_1$, both t_2 and t_3 are enabled. If t_2 fires, the new marking, say $\mathcal{M}_2 = (1, 1, 1, 1)$. If t_3 fires, the new marking, say $\mathcal{M}_3 = (1, 2, 0, 1)$.

Modelling Power of Petri Nets

The typical characteristics exhibited by the activities in a dynamic event-driven system such as concurrency, conflict, synchronization and parallelism can be modelled effectively by Petri nets.

Causality: The causal relationships between the occurrences of events can also be extracted from nets. For example, in Figure 2.6(a), transition t_2 can fire only after the firing of t_1. This imposes the precedence constraint "t_2 after t_1". Such precedence constraints are typical of the execution of the parts in a dynamic system.
Conflict: Conflict describes how the occurrence of one event can inhibit the occurrence of another in a marking. Transitions t_1 and t_2 are in conflict in Figure 2.6(b). The resulting conflict may be resolved in a purely non-deterministic way or in a probabilistic way, by assigning appropriate probabilities to the conflicting transitions.
Concurrency or Parallelism: Two transitions are parallel at a given marking if they can be fired at the same time, i.e., simultaneously. Parallel activities or concurrency can be easily expressed in terms of Petri nets. For example, in the Petri net shown in Figure 2.6(c), the parallel or concurrent activities represented by transitions t_2 and t_3 begin at the firing of transition t_l. In general, two transitions are said to be concurrent if they are causally independent, i.e., one transition may fire before or after or in parallel with the other.

Synchronization: It is quite normal in a dynamic system that an event requires multiple resources. The resulting synchronization of resources can be captured by transitions of the type shown in Figure 2.6(d). Here, t_1 is enabled only when each of p_1 and p_2 has a token. The presence of a token into each of the two places could be the result of a possibly complex sequence of operations elsewhere in the rest of the Petri net model. Essentially, transition t_1 models the join operation.
Mutually exclusive: Two events are mutually exclusive if they cannot be performed at the same time. Figure 2.6(e) shows this structure.
Priorities: The classical Petri nets discussed so far have no mechanism to represent priorities. Such a modelling power can be achieved by introducing an inhibitor arc. The inhibitor arc connects an input place to a transition, and is pictorially represented by an arc terminated with a small circle. The presence of an inhibitor arc connecting an input place to a transition changes the transition enabling conditions. In the presence of the inhibitor arc, a transition is regarded as enabled if each

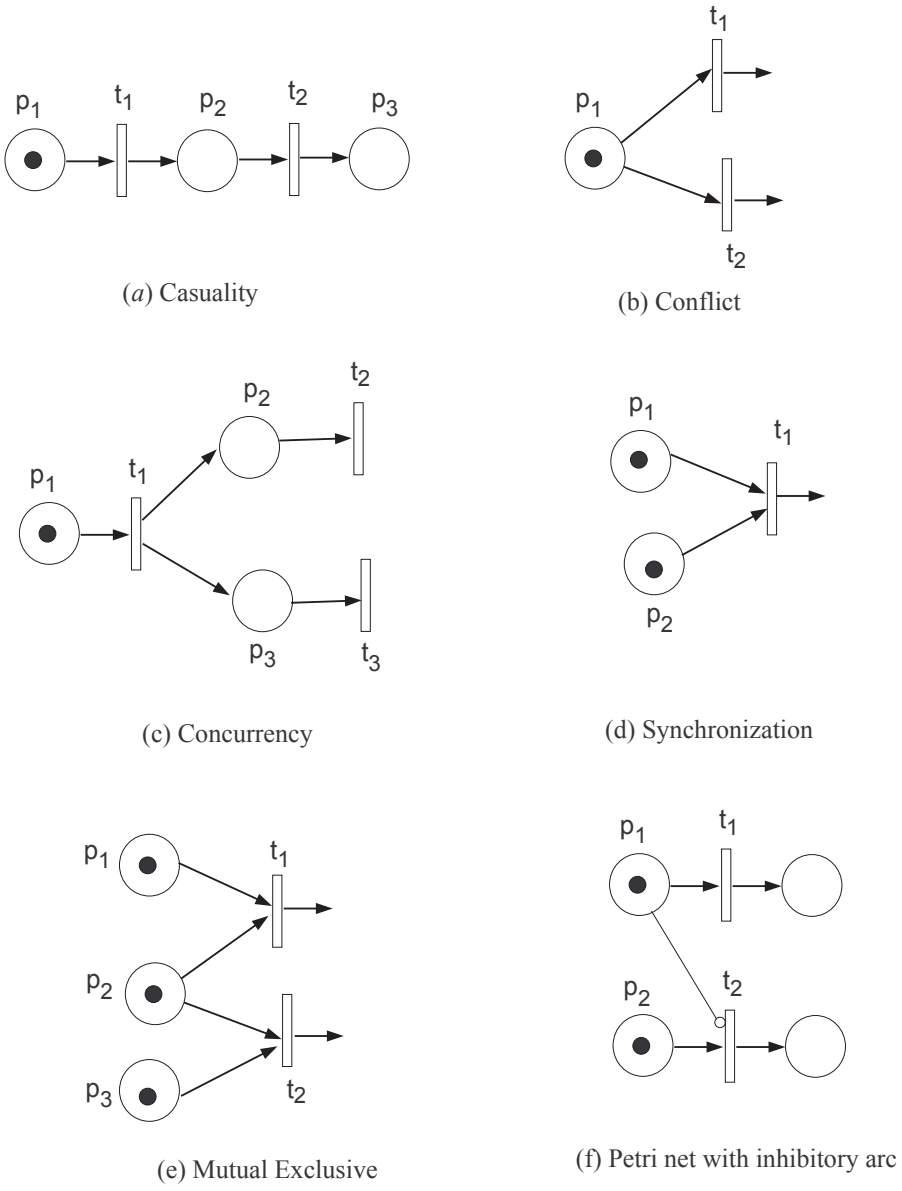


Figure 2.6: Petri net primitives to represent system features.

input place, connected to the transition by a normal arc (an arc terminated with an arrow), contains at least the number of tokens equal to the weight of the arc, and no tokens are present on each input place connected to the transition by the inhibitor arc. The transition firing rule is the same for normally connected places. The firing, however, does not change the marking in the inhibitor arc connected places. A Petri

net with an inhibitor arc is shown in Figure 2.6(f). t_1 is enabled if p_1 contains a token, while t_2 is enabled if p_2 contains a token and p_1 has no token. This gives priority to t_1 over t_2. The introduction of inhibitor arcs adds the ability to test "zero" (i.e., absence of tokens in a place) and increases the modelling power of Petri nets to the level of Turing machines [84].

2.7.2 Petri Net Properties

As a mathematical tool, Petri nets possess a number of properties. These properties, when interpreted in the context of the modelled system, allow the system designer to identify the presence or absence of the application domain specific functional properties of the system under design. Two types of properties can be distinguished, behavioural and structural ones. Here we provide an overview of some of the most important properties. Refer to [76, 84] for details.

Behavioural Properties

The behavioural properties are those which depend on the initial state or marking of a Petri net. They are *reachability*, *safeness*, and *liveness*.

Reachability: In order to find out whether the modelled system can reach a specific state as a result of a required functional behaviour, it is necessary to find such a transition firing sequence which would transform a marking $\mathcal{M}_0$ to $\mathcal{M}_n$, where $\mathcal{M}_n$ represents the specific state. Reachability is a fundamental basis for studying the dynamic properties of any system. A marking $\mathcal{M}_n$ is reachable from the initial marking $\mathcal{M}_0$ if there is a sequence of firings that transforms $\mathcal{M}_0$ to $\mathcal{M}_n$. Furthermore, the firing of a sequence of transitions (ϑ) is defined as $\vartheta = t_1 t_2 \dots t_n$ such that $\mathcal{M}_0[t_1\rangle\mathcal{M}_1[t_2\rangle \dots [t_n\rangle\mathcal{M}_n$ which is abbreviated as $\mathcal{M}_0[\vartheta\rangle\mathcal{M}_n$. A marking $\mathcal{M}$ is reachable if there exists a firing sequence ϑ such that $\mathcal{M}_0[\vartheta\rangle\mathcal{M}$. The set of all possible markings reachable from $\mathcal{M}_0$ in a net is denoted by $R(\mathcal{M}_0)$.

A marking $\mathcal{M}$ in a Petri net is said to be *coverable* if there exists a marking $\mathcal{M}'$ in $R(\mathcal{M}_0)$ such that $\mathcal{M}'(p) \leq \mathcal{M}(p)$ for each p in the net.

The reachability problem for Petri net is the problem of finding if a marking $\mathcal{M}$ is reachable from the initial marking $\mathcal{M}_0$. In some applications, one may be interested in the markings of a subset of places and not care about the rest of places in the net. This leads to a submarking reachability problem which is the problem of finding if $\mathcal{M}' \in R(\mathcal{M}_0)$, where $\mathcal{M}'$ is any marking whose restriction to a given subset of places agrees with that of a given marking $\mathcal{M}$.

Safeness: In a Petri net, places are often used to represent information storage areas in communication and computer systems. It is important to be able to determine whether proposed control strategies prevent from the overflows of these storage areas. The Petri net property which helps to identify the existence of overflows in the modelled system is the concept of *boundedness*.

A place p is said to be *k-bounded* if the number of tokens in p is always less than or equal to k (k is a nonnegative integer number) for every marking $\mathcal{M}$ reachable from the initial marking $\mathcal{M}_0$, i.e., $\mathcal{M} \in R(\mathcal{M}_0)$. It is safe if it is 1-bounded. A Petri net (N) is k-bounded (safe) if each place in P is k-bounded (safe).

Liveness: Formally a Petri net with a given marking is said to be in *deadlock* if and only if no transition is enabled in the marking. A Petri net where no deadlock can occur starting from a given marking is said to be live. This implies that for any reachable marking $\mathcal{M}$, it is ultimately possible to fire any transition in the net by progressing through some firing sequence.

Structural Properties

The structural properties, on the other hand, do not depend on the initial marking of a Petri net. They depend on the topology or net structure of a Petri net. Thus, these properties may be characterised in terms of the incidence matrix. The important

structural properties of a Petri net include *conservativeness, consistency, traps* and *siphons.*

Conservativeness: A Petri net is strictly conservative if the total number of tokens is constant in each marking of $R(\mathcal{M}_0)$. A subset of places form a *place-invariant* (or *P-invariant*) if it is strictly conservative.

Consistency: A Petri net $\mathcal{N}$ is said to be (partially) consistent if there exists a firing sequence ϑ from $\mathcal{M}_0$ back to $\mathcal{M}_0$, such that every (some) transition occurs at least once in ϑ. The set of all transitions in ϑ form *transition-invariant* (or *T-invariant*).

Siphons and Traps: A siphon is a set R of places such that $\forall p \in R$, $I(p) \subseteq O(p)$ while conversely, a trap is a set S of places such that $\forall p \in S$, $O(p) \subseteq I(p)$, where $I(p)$ ($O(p)$) denotes the set of input (respectively, output) transitions of place p. i.e. every transition having an output place in R has an input place in R and every transition having an input place in S has an output place in S.

A relevant property of a siphon is that once it is token-free under some marking $\mathcal{M}$ (it is not marked by $\mathcal{M}$), then, it remains token-free under all reachable markings from $\mathcal{M}$. Conversely, a relevant property of a trap is that once one of its places is marked under some marking, then the trap remains marked under all reachable markings from $\mathcal{M}$. As the definitions of siphons and traps are symmetrical, the properties of siphons also hold also for traps.

A *minimal siphon* is a siphon which doesn't contain any other siphon. Whereas, basis siphon is a siphon, that couldn't be obtained by the union of other siphons.

2.7.3 Analysis of Petri Nets

The success of any model depends on two factors: its modelling power and its decision power. Modelling power refers to the ability to correctly represent the system to be modelled; decision power refers to the ability to analyse the model and determine properties of the modelled system. The modelling power of Petri nets has been

examined in the previous sections, and in this section we take into consideration the analysis techniques of Petri nets.

There are two common approaches to Petri net analysis: *reachability analysis* and the *matrix-equation approach.* The first approach involves the enumeration of all reachable markings and is very useful in studying the behavioural properties of the system. The matrix equations technique is powerful for analysing the structural properties of the system.

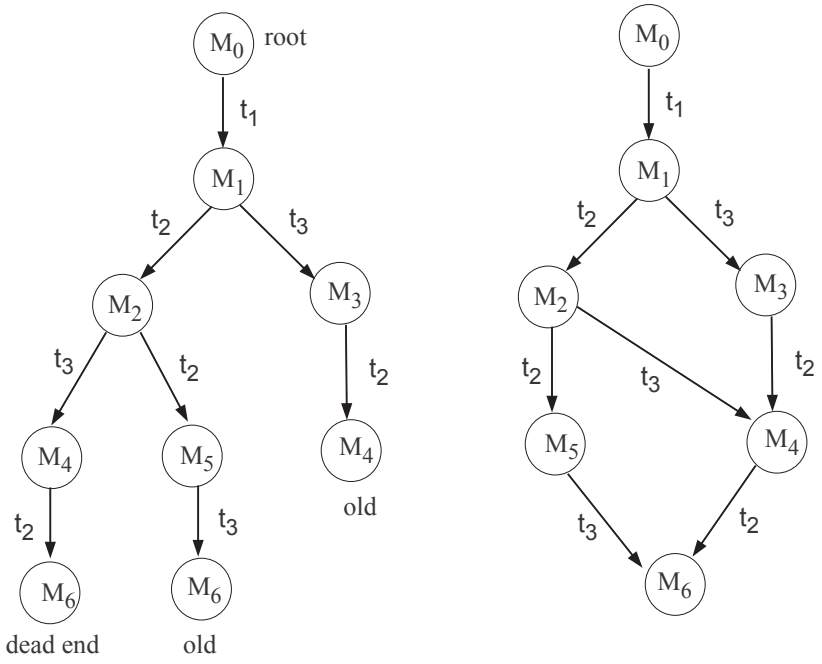
Reachability Analysis: Reachability analysis is conducted through the construction of the coverability tree. A Coverability tree is a tree representation of all possible markings with initial marking as the root node and nodes as the markings reachable from $\mathcal{M}_0$ and arcs represent the transition firing. Given a Petri net $\mathcal{N}$, from its initial marking $\mathcal{M}_0$, we can obtain as many "new" markings as the number of the enabled transitions. From each new marking, we can again reach more markings. Repeating the procedure over and over results in a tree representation of the markings.

The above tree representation, however, will grow infinitely large if the net is unbounded. To keep the tree finite, we introduce a special symbol ω, which can be thought of as "infinity". It has the properties that for each integer n, $\omega > n$, $\omega + n = \omega$ and $\omega \geq \omega$.

For a bounded Petri net, the coverability tree is called the reachability tree since it contains all possible reachable markings. It defines a net's state space (i.e., the set of reachable states). Merging the same nodes in a reachability tree results in a reachability graph.

Consider the Petri net shown in Figure 2.5(a). All reachable markings are: $\mathcal{M}_0 = (3,0,0,0)$, $\mathcal{M}_1 = (1,2,1,0)$, $\mathcal{M}_2 = (1,1,1,1)$, $\mathcal{M}_3 = (1,2,0,1)$, $\mathcal{M}_4 = (1,1,0,2)$, $\mathcal{M}_5 = (1,0,1,2)$, and $\mathcal{M}_6 = (1,0,0,3)$. The reachability tree of this Petri net is shown in Figure 2.7(a), and the reachability graph is shown in Figure 2.7(b).

Matrix Equation Approach: Structural properties of Petri nets are those that depend only on their topological structure and are independent of the initial marking. Thus,



(*a*) Reachability tree (b) Reachability graph

Figure 2.7: Reachability analysis.

these properties may be characterised in terms of the incidence matrix A. For a Petri net $\mathcal{N}$ with n transitions and m places, the incidence matrix $A = [a_{ij}]$ is an $n \times m$ matrix of integers and its typical entry is given by

$a_{ij} = {a_{ij}}^{+} - {a_{ij}}^{-}$

where ${a_{ij}}^{+} = W(t_i, p_j)$ is the weight of the arc from transition t_i to its output place p_j and ${a_{ij}}^{-} = W(p_j, t_i)$ is the weight of the arc to transition t_i from its input place p_j.

It is easy to see from the transition firing rule that ${a_{ij}}^{-}, {a_{ij}}^{+}, a_{ij}$ respectively represent the number of tokens removed, added, and changed in place p_j when transition t_i fires once. Transition t_i is enabled at a marking $\mathcal{M}$ if and only if

${a_{ij}}^{-} \leq \mathcal{M}(p_j)$, $j = 1, 2, \ldots, m$. In writing matrix equations, we write a marking $\mathcal{M}_k$ as an $m \times 1$ column vector. The jth entry of $\mathcal{M}_k$ denotes the number of tokens in place p_j immediately after the kth firing in some firing sequence. The kth firing or control vector u_k is an $n \times 1$ column vector of $n - 1$ zeroes and one nonzero entry, a 1 in the

ith position indicating that transition t_i fires at the kth firing. Since the ith row of the incidence matrix A denotes the change of the marking as the result of firing of transition t_i, we can write the following state equation for a Petri net:

$\mathcal{M}_k = \mathcal{M}_{k-1} + A^T u_k$, $k = 1, 2, \ldots$

Suppose that a destination marking $\mathcal{M}_n$ is reachable from $\mathcal{M}_0$ through a firing sequence $\{u_1, u_2, \ldots, u_n\}$.

Writing the state equation for $k = 1, 2, \ldots, d$ and summing them, we obtain

$\mathcal{M}_n = \mathcal{M}_0 + A^T \sum_{k=1}^{n} u_k$

which can be rewritten as

$A^T X = \Delta M$

where $\Delta\mathcal{M} = \mathcal{M}_n - \mathcal{M}_0$ and $X = \sum_{k=1}^{n} u_k$. Here X is an $n \times 1$ column vector of non negative integers and is called the firing count vector. The ith entry of X denotes the number of times that transition i must fire to transform $\mathcal{M}_0$ to $\mathcal{M}_n$.

The state equation is illustrated below, for the Petri net shown in Figure 2.5(a), where the transition t_1 fires to reach next marking $\mathcal{M}_1 = (1, 2, 1, 0)^T$ from $\mathcal{M}_0 = (3, 0, 0, 0)^T$.

$$\begin{pmatrix} 1 \\ 2 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -2 & 0 & 0 \\ 2 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

An n-vector X of positive (non negative) integers is T-invariant iff $A^T X = 0$, $X \neq 0$.

An m-vector Y is an P-invariant iff $\mathcal{M}^T Y = {\mathcal{M}_0}^T Y$ for any fixed initial marking $\mathcal{M}_0$ and any $\mathcal{M} \in R(\mathcal{M}_0)$.

2.7.4 High Level Petri Nets

Petri nets have been used to describe a wide range of systems since their invention in 1962. A problem with Petri nets is the explosion of the number of elements of their graphical form when they are used to describe complex systems. High-level Petri nets were developed to overcome this problem by introducing higher-level

concepts, such as the use of complex structured data as tokens, arc annotations, transition functions (guards) and time delays to places, transitions or arcs etc. The term high-level Petri net is used for many Petri net formalisms that extend the basic P/T net formalism; this includes coloured Petri nets, timed Petri nets, and all other extensions sketched in this subsection. In the use of Petri net for modelling real systems several authors have found convenient to use all or some of these concepts and introduce special constructs either for making the model representation more compact in a given application or for extending the modelling power of the Petri net formalism. Here we discuss some important structural components of the high level Petri nets.

Structural Components

Labelled tokens: The common characteristic of these models, usually referred to as high level Petri net, is that the position of any single token can be tracked in the Petri net. Two labelling techniques have been originally proposed: the technique of colouring tokens (coloured Petri net introduced by Jensen [54]) and the technique of assigning to each token a predicate (Predicate/Transition net introduced by Genrich and Lautenbach [27]).

Arc annotations: Arcs are inscribed with expressions which may comprise constants, variables (e.g., x, y) and function images (e.g., $f(x)$). The variables are typed. The expressions are evaluated by assigning values to each of the variables. When an arc's expression is evaluated, it must result in a collection of items taken from the type of the arc's place. The collection may have repetitions. The arcs are also inscribed with marking dependent expressions of various types, where the number of tokens removed from a place, or added to a place varies according to the marking of the Petri net [20].

Guard functions or Conditioning functions: More complex logical interactions between primitive elements of a Petri net can be considered by introducing logical

conditioning functions. Given a marking $\mathcal{M}$, a Petri net transition is enabled if, besides the normal enabling requirements (including inhibitor arcs and priorities), the conditioning function is true. The conditioning functions can be very effective in reducing the graphical complexity of a Petri net. The coloured Petri nets in [54] uses guard functions and arc annotations.

Time: The need for including timing variables in the models of various types of dynamic systems is apparent since these systems are real time in nature. There are many ways to introduce time into Petri net models. Time can be associated with places, transitions, or arcs. In most timed Petri net models, transitions determine time delays. In only a few models, time delays are determined by places and/or arcs. Independent of the choice where to put the delay (i.e., transitions, places, or arcs), several types of delays can be distinguished. Petri net models by authors such as Ramchandani [90], Sifakis [95] use deterministic delays, i.e., the delay assigned to a transition, place, or arc is fixed. Some authors such as Molloy [75] use stochastic delays, i.e. time variables are associated with transitions. These are the two most widely used timed Petri nets.

2.7.5 Petri Net Languages

In 1976, Hack [41] published a report on Petri net languages where he stated that in many applications of Petri nets it is the set of firing sequences generated by the net that is of prime importance. At this time it was proposed to treat Petri nets like an automaton whose states are the markings of the Petri net, and whose state-transition function expresses how and when transitions of the Petri net can fire. This report was the start of an extensive research effort in Petri net languages, which resulted in the definition of a wide range of Petri net language families each having their own properties.

This subsection introduces the basic concepts of Petri net languages, for a more

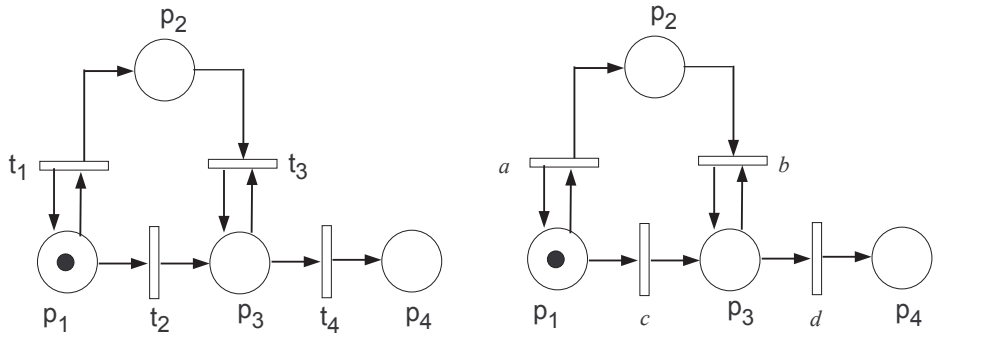
elaborate discussion the reader is referred to [41, 83, 84]. Basically, a Petri Net language is generated by a labelled Petri net $\mathcal{K} = (V, \mathcal{N}, \zeta, F)$ where

- V is an alphabet.
- $\mathcal{N} = (P, T, F, W, \mathcal{M}_0)$ is a P/T Net.
- $\zeta : T \to V \cup \{\lambda\}$ defines the symbol-wise labelling for every transition. A transition $t \in T$ is called λ-transition, if $\zeta(t) = \lambda$.
- H is a finite set of final markings.

Definition 2.11. *Let $\mathcal{K} = (V, \mathcal{N}, \zeta, H)$, $\mathcal{N} = (P, T, F, W, \mathcal{M}_0)$, be a labelled Petri net. For any two markings $\mathcal{M}$, $\mathcal{M}'$ and transition $t \in T$, we write $\mathcal{M} \overset{\zeta(t)}{\Longrightarrow} \mathcal{M}'$ if $\mathcal{M}[t\rangle\mathcal{M}'$. This is recursively generalized to (labelled) firing sequences: Let $\vartheta \in T^*$ and $t \in T$ and $\mathcal{M}_1$ and $\mathcal{M}_3$ are any two markings then $\mathcal{M}_1 \overset{\zeta(\vartheta)\zeta(t)}{\Longrightarrow} \mathcal{M}_3$ iff $\exists$ a marking $\mathcal{M}_2$ such that $\mathcal{M}_1[\vartheta\rangle\mathcal{M}_2[t\rangle\mathcal{M}_3$.*
In addition, let $\mathcal{M} \overset{\lambda}{\Longrightarrow} \mathcal{M}$ for any marking $\mathcal{M}$.

Four types of Petri net languages have been defined in terms of the definition of final markings [83]:

- According to Hack, the *L-type* Petri net language is
  $L(\mathcal{K}) = \{\zeta(\vartheta) \in V^* \mid \vartheta \in T^*, \exists \mathcal{M} \in H, \mathcal{M}_0[\vartheta\rangle\mathcal{M}\}$
  the *T-type* Petri net language is:
  $T(\mathcal{K}) = \{\zeta(\vartheta) \in V^* \mid \vartheta \in T^*, \mathcal{M}_0[\vartheta\rangle\mathcal{M}, \forall t \in T, \mathcal{M}\overline{[t\rangle}\}$
  the *P-type* Petri net language is:
  $P(\mathcal{K}) = \{\zeta(\vartheta) \in V^* \mid \vartheta \in T^*, \mathcal{M}_0[\vartheta\rangle\}$
  the *G-type* Petri net language is:
  $G(\mathcal{K}) = \{\zeta(\vartheta) \in V^* \mid \vartheta \in T^*, \mathcal{M}_0[\vartheta\rangle\mathcal{M}, \mathcal{M}' \geq \mathcal{M}$ for some $\mathcal{M}' \in H\}$
  $L(\mathcal{K})$ is called λ-free labelled, iff $\zeta(t) \neq \lambda$ for each $t \in T$.



(a) A Petri net with initial marking (b) A labelled Petri net corresponding to Figure (a)

Figure 2.8: A labelled Petri net.

As an example, consider the Petri net of Figure 2.8(b) which is a labelled version of the net of Figure 2.8(a).

Labelling function ζ for the transitions is defined as

$\zeta(t_1) = a$, $\zeta(t_3) = b$,

$\zeta(t_2) = c$, $\zeta(t_4) = d$.

For a final state set $H = \{(0,0,1,0)\}$,

the L-type language is $\{a^n c b^n \mid n \geq 0\}$,

the G-type language is $\{a^m c b^n \mid m \geq n \geq 0\}$,

the T-type language is $\{a^m c b^n d \mid m \geq n \geq 0\}$, and

the P-type language is $\{a^m \mid m \geq 0\} \cup$

$\{a^m c b^n \mid m \geq n \geq 0\} \cup \{a^m c b^n d \mid m \geq n \geq 0\}$.

2.7.6 Parallel Petri Net Semantics

The execution mode considered so far in the Petri nets is the sequential mode. This mode leads to interleaving semantics of Petri nets. In this mode only one of the enabled transitions is executed in each step. Classes of languages that reflect an interleaving semantics are usually defined by Petri nets as sets of sequences of labelled or unlabelled transitions as discussed in the previous subsection. Petri nets are widely

used as a model of concurrency, which allows to represent the occurrence of independent events. They can be as well a model of parallelism, where the simultaneity of the events is more important, when we consider their *step sequence semantics* in which an execution is represented by a sequence of steps each of them being the simultaneous occurrences of transitions [14, 53].

Steps in Petri nets are sets of transitions that fire independently and parallel at the same time. The change to the marking of the net when a step occurs is given by the sum of all the changes that occur for each transition. As in [53], if every transition occurs only once, like in elementary net systems, thus having single usage within the step, then we use the naming step. A multiset of transition, on the other hand, may contain more than one occurrence of a transition. In this case, a transition can fire several times in one step. Since such multisets of transitions can be seen as the sum of several single steps, they will shortly be called multi-steps.

A step of transitions U is a maximal step at a marking $\mathcal{M}$, if $\mathcal{M}[U\rangle$ and there is no transition t' such that $\mathcal{M}[U + t'\rangle$. A Petri net system $\mathcal{N}$ with *maximal strategy* is such that for each markings $\mathcal{M}$ and $\mathcal{M}'$ if there is a step U such that $\mathcal{M}[U\rangle\mathcal{M}'$, then U is a maximal step and we write as $\mathcal{M}[U\rangle_m\mathcal{M}'$. In [13], it is proved that P/T systems with maximal strategy can perform the test for zero and so the computational power is extended up to the power of Turing machines.

The notions of steps and maximal steps are recursively generalized to step sequences and maximal step sequences. A (maximal) step sequence ρ is *halting* if $\mathcal{M}_0[\rho\rangle\mathcal{M}_n$ and no non-empty (maximal) step is fireable at $\mathcal{M}_n$. A computation of a Petri net $\mathcal{N}$ is a halting (maximal) step sequences starting from the initial marking and every marking appearing in such a sequence is called reachable.

Burkhard [14] has defined languages using Petri net steps by writing down all permutations of the transitions that form such a step. He thereby defines the Petri net language in an interleaving semantics, which does not directly reflect the parallel use of transition firings.

In [53], authors allowed only (multi-)sets of transitions to form a (multi-) step, if they all share the same label. In a sequence of (multi-) steps, each of them contributes its label once to the generated word. Through different firing modes that allow multiple use of transitions in a single multi-step, they obtained a hierarchy of families of languages.

In the thesis, we consider parallel Petri net semantics for translating SN P systems into Petri nets. Instead of labelling transitions, we label the (maximal) steps. The halting sequences of (maximal) steps are considered for the languages generated by Petri nets.

2.7.7 Simulation of Petri Nets

Petri nets are one of well established tools in both theoretical analysis and practical modelling of concurrent systems as well as approximate reasoning. However, practical usage of Petri nets is increased by the availability of large number of computer tools which would allow to handle large and complex nets in a comfortable way. Four things are essential for modelling and analyzing by means of Petri nets - capability to represent the required features of the systems under design, a good editor, a simulator and a powerful analysis engine. Analysis shows the presence of undesirable properties. Petri net simulation is indeed a convenient and straightforward yet effective approach for engineers to validate the desired properties of a discrete event system. A list of Petri net simulation tools along with feature descriptions can be found in the Petri nets world website: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/. The *Petri Net Markup Language* (PNML) is a standard XML based interchange format for Petri nets. In order to support different versions of Petri nets and, in particular, future versions of Petri nets, PNML allows the definition of Petri net types. It helps to easily exchange Petri nets between different tools. This would allow Petri net tool users in geographically distributed locations to take advantage of newly developed facilities on other tools, for example, for

analysis, simulation or implementation.

2.7.8 An Overview of PNetLab

PNetLab 4.0 is a Java based tool that provides interactive simulation, analysis, and supervision of Petri nets. The tool allows the Petri nets in Petri Net Markup Language transfer format. The simulation engine has been developed in C++, it works in cooperation with a graphical user interface and provides interactive simulation with graphical animation of the model and movement of the tokens, step-by-step and off-line simulations, forward and backward time progression. It allows drawing and simulation of coloured Petri nets, P/T nets, timed/untimed models by means of a Java graphical user interface.

For P/T nets without guard, the tool provides T-invariants, P-invariants, siphons, traps, pre-incidence, post-incidence, and incidence matrices, and coverability tree. PNetLab allows the integration of a PN/CPN model with a standard C/C++ control algorithm thus allowing closed-loop analysis and simulation of supervised systems. It is also interfaced with Xpress by Dask Optimization, a tool of linear programming. It allows to build user defined arc (guard) functions by combining the built-in functions and several mathematical functions in accordance with the C/C++ syntax.

The syntax of different built-in guard functions used in the thesis are as follows:

- **all(i)**: It selects all the tokens of the place p_i.
- **ntoken(i)**: It returns the number of tokens of the place p_i.

PNetLab allows the firing of multiple transitions in a single step and resolves conflicts. It manages conflicts by using the following resolution policies:

1. Predefined Scheduling order: PNetLab assigns a static priority to the transition in conflict, based on the order in which they have been drawn;

2. Same firing rate: transition in conflict relation have the same firing probability;

3. Stochastic firing rate: transition in conflict relation have a firing probability defined a priori by the user;

In the thesis, we used the same firing rate option to resolve the conflicts during the execution of the Petri net. In each step, the tool allows only one transition to fire from each input place which makes the simulation of SN P systems easier (since SN P system allows only one rule to fire from each neuron in each step). A detailed manual about PNetLab can be found in [3].

2.8 P systems and Petri Nets

In order to study the behavioural properties of P systems, several authors proposed procedures to model them with Petri nets. Since multiset calculus is basic for membrane systems and also for computing the token distribution in Petri nets [16], some connections have already been established. Some authors have proposed to interpret reaction rules of membrane systems using Petri net transitions, e.g., [7, 89]

A key structural notion is that of a membrane by which a system is divided into compartments where chemical reactions can take place. These reactions transform multisets of objects present in the compartments into new objects, possibly transferring objects to neighbouring compartments, including the environment. Consequently, the behavioural aspects of membrane systems are based on sets of reaction rules defined for each compartment. Places together with their markings indicate the local availability of resources and thus can be used to represent objects in specific compartments, whereas transitions are actions which can occur depending on local conditions related to the availability of resources and thus can be used to represent reaction rules associated with specific compartments. When a transition occurs it consumes resources from its input places and produces items in its output

places thus mimicking the effect of a reaction rule. Every object can be represented as a place in the P/T net, and the number of tokens in this place denotes the number of occurrences of this object.

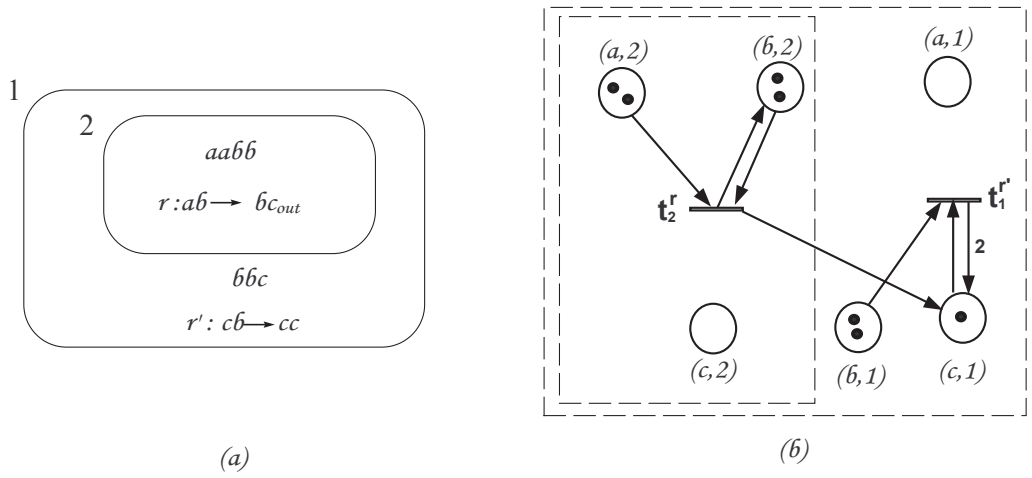


Figure 2.9: A membrane system (a), and the corresponding Petri net (b).

The basic idea of modelling a membrane system using a Petri net can be explained through an example shown in Figure 2.9. The system depicted there consists of two nested membranes (the inner membrane 2 and the outer membrane 1), two rules (rule r associated with the compartment c_2 inside the inner membrane, and rule r' associated with the compartment c_1 surrounded by membrane 1, i.e., in-between the two membranes), and three symbols denoting objects (a, b, and c). Initially, the compartment c_2 contains two copies of both a and b, and c_1 contains two copies of b and a single copy of c. To model this membrane system using a Petri net, we introduce a separate place (x, j) for each kind of object x and compartment c_j. As usual, places are drawn as circles with the number of the currently associated resources represented as tokens (small black dots). Every rule can be represented by a transition. For example, in compartment c_2, the rule $r : ab \to bc_{out}$, can be described by a transition t_2^r. Transitions are connected to places by weighted directed arcs, and if no weight is shown it is by default equal to 1. If the transformation described

by a rule r of compartment c_j consumes k copies of object x from compartment c_j, then we introduce a k weighted arc from place (x, j) to transition t_i^r, and similarly for objects produced by transformations.

Finally, assuming that initially compartment c_j contained n copies of object x, we introduce n tokens into place (x, j). The resulting Petri net is depicted in Figure 2.9(b).

In order to represent the compartmentisation of membrane systems some authors introduced a novel extension of the basic net model of P/T nets, by adding the notion of located transitions and locally maximally concurrent executions of co-located transitions [60]. In these Petri nets, called P/T nets with localities (PTL-nets), each transition has a location, similar to the distribution of the reaction rules over the compartments in a membrane system. The exact mechanism for achieving this is to introduce a partition of the set of all transitions, using a locality mapping function. Intuitively, two transitions for which the locality function returns the same value will be co-located. It has been shown how (sequences of) computation steps of membrane systems are faithfully reflected in the maximally concurrent step sequence semantics of their corresponding PTL-nets. In the P systems area maximal step semantics correspond to maximal multi-step semantics discussed in the thesis.

Note that for maximal concurrency in a P/T net, localities are not relevant, as the net supports the local aspects of resources consumed and produced by transitions. Localities are primarily a modelling tool in that co-located transitions correspond to reaction rules in a single compartment and, e.g., allow to identify the active parts of a system in the course of a computation. However, transitions with associated localities can be used to restrict synchronicity to certain locations within a system: within each clock tick, for each currently active locality, as many transitions belonging to this locality as possible are executed. Thus the PTL-net model and its locally maximal concurrent step semantics make it possible to investigate membrane systems working subject to the natural assumption that synchronicity is

restricted to the compartments of the system as delineated by the membranes.

To model the membrane systems with inhibitors and promoters of reactions [6], PTL-nets are extended with inhibitor and activator arcs [57]. In [58], dynamic membrane systems with dissolving and thickening rules[29] are modelled directly and soundly in Petri nets with localities and supporting activator and inhibitor arcs [58]. Inhibitor arcs are the arcs, where the enabling of an action (transition) can depend on some specific local states (or places) being unmarked and activator (or read) arcs are the arcs, where the enabling of an action (transition) can depend on some specific local states (or places) being marked by more tokens than just the number of those consumed when the transition is fired. Range arcs combine (and subsume) the distinctive features of inhibitor and activator arcs, and each such arc provides a means of specifying a range (a finite or infinite interval of non negative integers) for the number of tokens in a place which makes enabling of a given transition possible [56].

The major limitations of these translations is that these new variants of Petri nets lack of tools to simulate and study the behavioural properties of P systems.

With these prerequisites we now move on to the main portions of the thesis.

Chapter 3

Computability of Spiking Neural P Systems with Anti-Spikes

Because of the use of two types of objects, spiking neural P systems with anti-spikes can encode the binary digits in a natural way and hence represent the formal models more efficiently and naturally. They can also generate more languages than the standard SN P systems. This chapter deals with the computing power of spiking neural P system with anti-spikes. We discuss the efficiency of the systems as language generators in Section 3.2. It is demonstrated in Section 3.3 that, as transducers, spiking neural P systems with anti-spikes can simulate any Boolean circuit and also computing devices such as finite automata. In Section 3.4, we investigate how the use of anti-spikes increase the efficiency to solve the satisfiability problem in a non-deterministic way.

3.1 Introduction

The power of different variants of SN P systems as language generators are investigated in [18, 19, 34]. It was shown in [18] that some finite languages cannot be

generated using simple SN P systems but it was proved in [19] that SN P systems with extended rules can generate the finite languages. SN PA system uses standard rules, adding one symbol at a time, but allows non-determinism between its rules like $a^c \rightarrow a$ and $a^c \rightarrow \overline{a}$, thus helps to generate languages that cannot be generated by simple SN P systems. In this chapter we address the power of SN PA systems as language generators, in particular, by considering bounded SN PA systems and comparing the languages generated with the results obtained in [48] for standard SN P systems.

Standard spiking neural P systems are used to simulate arithmetic and logic operations where the presence of spike is encoded as 1 and absence of spike as 0 and the negative integers were not considered [73]. The ability of SN P systems to efficiently simulate Boolean circuits are studied [52], since this simulation, enriched with some “memory modules" (given in the form of some SN P sub-systems), may constitute an alternative proof of the computational completeness of the model. The Boolean value 1 is encoded in the SN P system by two spikes, hence a^2, while 0 is encoded as one spike. As the system has only one input neuron, the number of spikes equal to the sum of the inputs, is introduced into the neuron. For example to compute the logical AND or OR operation between 1 and 0 (or 0 and 1) three spikes (two spikes for 1 and one spike for 0) are introduced into the input neuron and four spikes are introduced for the case 11.

In this chapter we use SN P systems with anti-spikes to simulate logic gates, with the anti-spikes and spikes encoding the Boolean values 0 and 1 in the natural way. We design SN P systems with anti-spikes simulating the operations of AND, OR, NOT, NAND and NOR gates. The output of the system is 0 (hence false) if the output neuron sends out an anti-spike and 1 (true) if a spike is sent to the environment. Hence we present two ways to simulate any Boolean circuit, one is using fundamental gates and other using universal gates. Furthermore they are used to simulate computational devices such as finite state transducers.

Using SN P system with anti-spikes, we can perform the operations on negative numbers also. The input to the systems is a binary sequence of spikes and anti-spikes which encodes the digits 1 and 0 respectively, of a binary number. They can represent the negative numbers in 2's complement form, thereby simulating the arithmetic operations on negative numbers. In this chapter we also simulate three arithmetic operations - 2's complement, addition and subtraction.

Finally we show how they can be used for solving the SAT problem. In [66], a uniform solution to the SAT (in CNF, with n variables and m clauses) is provided using standard SN P systems without delay having $3n^2 + 8m + 5$ neurons, providing the solution in a number of steps which is linear in the number of variables. Two bits were used to code each literal of a clause, hence the computation cannot end in less than $2n$ steps. Here we use only one bit to code each literal of clause C_j. 1 indicates the case when x_i appears in C_j, 0 indicates the case when $\neg x_i$ appears in C_j and λ (empty) indicates the absence of x_i in the clause C_j. So n bits are needed to code any clause. Using SN PA systems the number of steps can be reduced to half. These systems only requires $3m + 2$ neurons.

3.2 SN PA Systems as Language Generators

The following observations show that some finite languages and regular languages which cannot be generated using simple bounded SN P systems proved in [18] can be generated using bounded SN PA systems with one neuron. Here $B = \{0, 1\}$ is the binary set. B^+ is the set of all binary strings formed using the alphabet B.

We recall that complexity of an SN PA system Π is described as $LSNPA_m(rule_k, cons_{p_1,p_2}, forg_{q_1,q_2})$, the family of languages $L(\Pi)$, generated by systems Π with at most m neurons, each neuron having at most k rules, each of the spiking rules consuming at most p_1 spikes and p_2 anti-spikes and each forgetting rule removing at most q_1 spikes and q_2 anti-spikes. As usual a parameter m, k, p_1, p_2, q_1, q_2

is replaced with $*$ if it is not bounded. If the underlying SN PA systems are finite, we denote the corresponding families of languages by $LFSNPA_m(rule_k, cons_{p_1,p_2}, forg_{q_1,q_2})$.

3.2.1 Finite Binary Languages

Observation 1 *The simplest language $L_{0,1} = \{0,1\}$ is generated by bounded SN P system with anti-spikes.*

SN PA system Π_1=($\{a, \overline{a}\}$, σ_1=$(1, \{a \to a, a \to \overline{a}\}), \emptyset, 1)$ generates the set $L_{0,1} = \{0,1\}$.

Observation 2 *Languages of the form $L_{k,j} = \{0^k, 10^j\}$, for $k \geq 1$, $j \geq 0$ can be generated by bounded SN PA system.*

We give an SN PA system Π_2 generating $L_{k,j} = \{0^k, 10^j\}$. Π_2=($\{a, \overline{a}\}$, σ_1=$(j+k, R_2), \emptyset, 1)$. where $R_2 = \{a^{j+k}/a^k \to a, a^{j+k}/a^{j+1} \to \overline{a}\} \cup \{a^l/a \to \overline{a} \mid 1 \leq l \leq j+k-1\}$.

Initially the output neuron σ_1 has $j+k$ spikes and there is a non-determinism between its rules $a^{j+k}/a^k \to a$ and $a^{j+k}/a^{j+1} \to \overline{a}$. If it uses the first rule, a spike is out and is left with j spikes. Then in the next j steps, the neuron uses the rule $a \to \overline{a}$, emitting j anti-spikes, thus generating the string 10^j. If the neuron uses later one then it emits an anti-spikes and is left with $k-1$ spikes, which are sent out as anti-spikes by the output neuron, thus generating the string 0^k. So $L(\Pi_2)$=$L_{k,j}$.

Theorem 3.1. *If $L = \{x\}$, $x \in B^+$, $|x|_1 = r \geq 0$, then $L \in LFSNPA_1(rule_{|x|}, cons_{1,0}, forg_{0,0})$, where $|x|$ is the length of the string x and $|x|_1$ is the number of occurrences of symbol 1 in x.*

Proof. Let us consider the string $x = 0^{n_1}10^{n_2}\ldots0^{n_r}10^{n_{r+1}}$, for $n_j \geq 0$, $1 \leq j \leq r+1$ (if $x = 0^{n_1}$, then $r = 0$). The SN P system from Figure 3.1 generates the string x. The output neuron initially contains $|x|$ spikes. The second set of rules $a^{|x|-(\sum_{i=1}^{j} n_i + j - 1)}/a \to a$ sends a spike(1) at $\sum_{i=1}^{j} n_i + j - 1$, $1 \leq j \leq r$ places where as the first set of rules allows an anti-spike to be sent out at other places. Depending upon the number of

spikes available, a unique rule is used in each step to generate either spike or anti-spike, resulting $|x|$ rules. In the case $r = 0$, the system cannot use the second set of

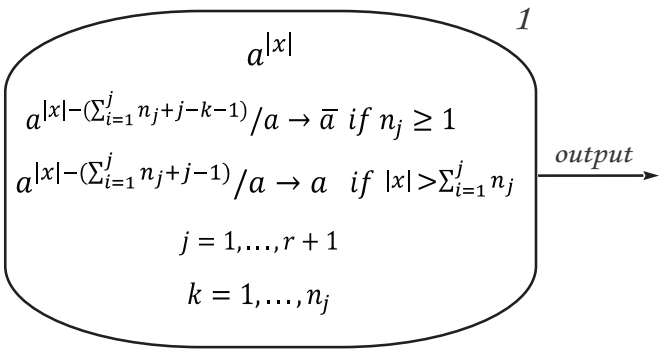


Figure 3.1: SN PA system generating a singleton set

rules as $|x| = n_1$. The first set of rules are used for $j = 1$ and $k = 1$ to n_1, outputting the string 0^{n_1}. □

Bounded SN P systems with standard rules cannot generate all binary finite languages, but with anti-spikes help in this respect.

Theorem 3.2. $LFSNPA_1(rule_*, cons_{*,*}, forg_{*,*}) = BFIN$, $BFIN$ *is the family of finite languages over binary alphabet.*

Proof. The inclusion $LFSNPA_1(rule_*, cons_{*,*}, forg_{*,*}) \subseteq BFIN$ can be easily proved. In each step, the number of spikes present in a system with only one neuron decreases by at least one, hence any computation lasts at most as many steps as the number of spikes/anti-spikes present in the system at the beginning. Thus, the generated strings have a bounded length.

To prove the opposite inclusion $BFIN \subseteq LFSNPA_1(rule_*, cons_{*,*}, forg_{*,*})$, let us take a finite language, $L = \{x_1, x_2, \ldots, x_m\} \subseteq B^*, m \geq 1$, and let $x_j = 0^{s_{j,1}}10^{s_{j,2}} \ldots 10^{s_{j,r_j+1}}$ for $r_j \geq 0$, $s_{j,l} \geq 0$, $1 \leq l \leq r_j + 1$, $1 \leq j \leq m$.
Let $| x_j | = n_j$, $1 \leq j \leq m$ and $\alpha_j = \sum_{i=1}^{j} n_i$, $1 \leq j \leq m$
An SN PA system which generates the language L is the following.

Π=($\{a, \overline{a}\}$, $\sigma_1, \emptyset, 1$), $\sigma_1 = (\alpha_m, R_1)$
R_1=($\{a^{\alpha_m}/a^{\alpha_m-(\alpha_j-1)} \rightarrow b \mid b = \overline{a}$ if $s_{j,1} \geq 1$ and $b = a$ if $s_{j,1} = 0, 1 \leq j \leq m\}$
$\cup\{a^{\alpha_j-1-(\sum_{i=1}^{l} s_{i,l}+l-k-1}/a \rightarrow \overline{a} \mid s_{j,1} \geq 2, s_{j,l} \geq 1, 2 \leq l \leq r_j+1, 1 \leq k \leq s_{j,l}, 1 \leq j \leq m\}$
$\cup\{a^{\alpha_j-1-(\sum_{i=1}^{l} s_{i,l}+l-1}/a \rightarrow a \mid 1 \leq l \leq r_j, 1 \leq j \leq m\} \cup\{a^{\alpha_j-1} \rightarrow \lambda \mid 2 \leq j \leq m\})$.

Initially, only a rule $a^{\alpha_m}/a^{\alpha_m-(\alpha_j-1)} \rightarrow b$ can be used, and in this way it non-deterministically choose the string x_j to generate and output spike/anti-spike depending on the first bit of x_j. The neuron is left with $\alpha_j - 1$ spikes. The rules for generating the remaining bits are similar to rules of SN PA system in Theorem 3.1. After generating x_j, α_{j-1} spikes remain in the neuron, and are forgotten using the rule $a^{\alpha_{j-1}} \rightarrow \lambda$.

We observe that the rules which are used in the generation of a string x_j cannot be used in the generation of a string x_k with $k \neq j$. □

3.2.2 Regular Binary Languages

We now pass to investigating the relationships with the family of regular languages over the binary alphabet. It was proved in [48] that 0^*1 cannot be generated by any bounded SN P system. But with SN PA system we can generate the language 0^*1.

Observation 3. *The language* 0^*1 *can be generated by a bounded SN PA system.*

Consider the SN PA system Π_4, which is formally denoted as
Π_4=(O, σ_1, σ_2, *syn* , 2), with
σ_1 = (-1, $\{\overline{a} \rightarrow a\}$), σ_2 = (2, $\{a^2/a \rightarrow \overline{a}$, $a^2 \rightarrow a\}$), *syn*=$\{(1, 2), (2, 1)\}$.
The graphical representation of the SN PA system Π_4 is shown in Figure 2.4 of Chapter 2, which generates the language 0^*1.

Theorem 3.3. $LFSNPA_*(rule_*, cons_{*,*}, forg_{*,*}) = BREG$, *BREG is the family of regular binary languages.*

Proof. The inclusion $LFSNPA_*(rule_*, cons_{*,*}, forg_{*,*}) \subseteq BREG$ follows from the

fact that for each finite SN PA system, we can construct the corresponding transition diagram associated with the computations of the SN PA system and then interpret it as the transition diagram of a finite automaton (with an arc labelled by 1 when the output neuron sends a spike and labelled by 0 when the output neuron sends an anti-spike).

To prove the opposite inclusion that if $L \subseteq B^*$, $L \in BREG$, then $L \in LFSNPA_*(rule_*, cons_{*,*}, forg_{*,*})$, we consider the right-linear grammar $G = (N, T, S, P)$ such that $L = L(G)$ and having the following properties.

i. $N = \{A_1, A_2, \ldots, A_n\}$, $n \geq 1$ and $S = A_n$.

ii. The rules in P are of the form $A_i \to 0A_j \mid 1A_j \mid 0 \mid 1$ where $i, j \in \{1, 2, \ldots, n\}$.

We construct the following SN PA system:
Π=($\{a, \overline{a}\}$, σ_1, $\sigma_2, \ldots, \sigma_{n+1}$, ***syn*** , $n+1$), with
σ_i = (1, $\{a \to a, a \to \overline{a}\}$), $i = 1, 2, \ldots, n$,
σ_{n+1} = ($3n$, $\{a^{2n+i}/a^{n+i-j} \to b' \mid A_i \to bA_j \in P\} \cup \{a^{2n+i} \to b' \mid A_i \to b \in P\}$) where $b \in \{0, 1\}$ and $b' = a$ if $b = 1$ and $b' = \overline{a}$ if $b = 0$,
syn=$\{(1, n+1), (n+1, 1), (2, n+1), (n+1, 2), \ldots, (n, n+1), (n+1, n)\}$.
For easier understandability, the system is also given graphically in Figure 3.2. The output neuron σ_{n+1} fires in the first step by a rule $a^{2n-j} \to b'$ (or $a^{3n} \to b'$) associated with a rule $A_n \to bA_j$ (or $A_n \to b$) from P, produces either a spike or an anti-spike depending upon whether $b = 1$ or $b = 0$ and receives n spikes from its neighbouring n neurons. The neurons σ_1 to σ_n are meant to continuously load the neuron $n+1$ with n spikes, provided that they receive spike or an anti-spike from the output neuron.

Assume in some step t, the rule $a^{2n+i}/a^{n+i-j} \to b'$, for $A_i \to bA_j$ or $a^{2n+i} \to b'$ for $A_i \to b$ is used, for some $1 \leq i \leq n$, and n spikes are received from other neurons.

If the first rule is used, then $n + i - j$ spikes are consumed and $n + j$ spikes remain in the output neuron. Then in the step $t + 1$, we have $2n + j$ spikes in neuron

σ_{n+1}, and a rule for $A_j \to bA_l$ or $A_j \to b$ can be used. In this step also the output neuron receives n spikes from its neighbouring neurons. In this way, the computation continues, unless the second rule is used.

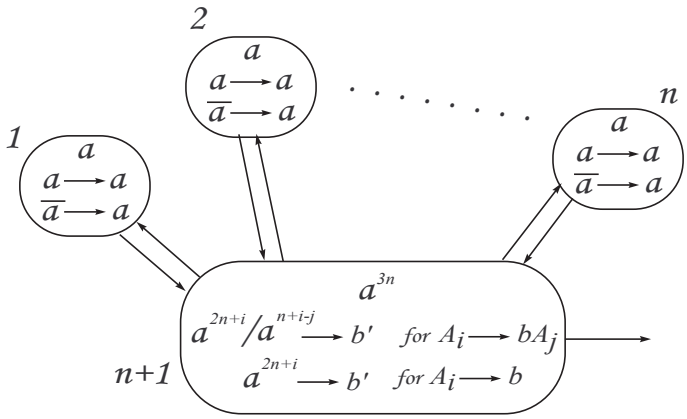


Figure 3.2: SN PA system generating binary regular languages

If the second rule is used, then all spikes of the output neuron are consumed sending a spike or an anti-spike to other n neurons and n spikes are received from them. Then in the next step the output neuron again receives n spikes, but no rule is used, so no spike is produced. So it stops loading the other n neurons and the computation halts. In this way, all strings in L can be generated. □

3.2.3 Going Beyond Regular Languages

The power of SN PA systems goes beyond the regular languages. We first illustrate this assertion with an example from Figure 3.3, that generates the language $L(\Pi) = \{0^n1^n \mid n \geq 1\}$; observe that the system is not finite due to the rule $(aa)^+a/a^2 \to \overline{a}$ in the σ_3 and the output is delayed for two steps.

The reader can check that in $n \geq 0$ steps when σ_2 uses the first rule $a^2/a \to a$, the neuron σ_3 accumulates $2n+2$ spikes and σ_1 sends a spikes to σ_4, which in turn sends a spike to output neuron, which uses its first rule and sends an anti-spike(0)

to environment. At any step $n \geq 1$, when the neuron σ_2 uses the rule $a^2 \rightarrow \overline{a}$, the spike from σ_1 and anti-spike from neuron σ_2 will annihilate each other in neuron σ_3, remaining again with $2n+2$ spikes. Neuron σ_2 receives a spike from neuron σ_5 where as σ_5 receives an anti-spike from neuron σ_2. In the same step spike from σ_1 is also sent to σ_4. In the $n+1$ step neurons σ_1 and σ_5 forget their anti-spikes received from σ_2. Neuron σ_4 sends a spike to neuron σ_6. Neuron σ_2 uses its third rule $a \rightarrow \overline{a}$ by sending an anti-spike to neurons σ_3 and σ_5. Neuron σ_3 is left with $2n+1$ spikes and σ_5 with an anti-spike which will be forgotten in the next step. In the $n+2$ step, the neuron σ_6 outputs a spike (that means total of $n+1$ spikes) and neuron σ_3 starts firing the as number spikes present becomes odd, and the rule $(aa)^+a/a^2 \rightarrow \overline{a}$ repeatedly used until one spike remains; this last spike is used by the second rule $a \rightarrow \overline{a}$. These $n+1$ anti-spikes are converted into spikes and sent to environment by the output neuron σ_6.

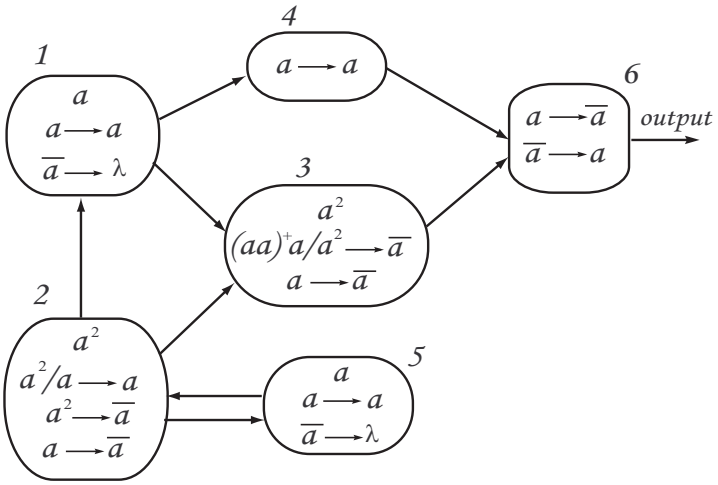


Figure 3.3: An SN P system with anti-spikes generating a context free language

Actually, much more complex languages can be generated by SN PA systems.

3.2.4 Going Beyond Context Free Languages

Actually, much more complex languages can be generated by SN PA systems. The previous construction can be extended to non-context free languages like $\{0^n1^n0^n/n \geq 1\}$.

Theorem 3.4. *Context sensitive languages can be generated by SN PA systems.*

Proof. The SN PA system from Figure 3.4 generates the language $\{0^n1^n0^n/n \geq 1\}$.

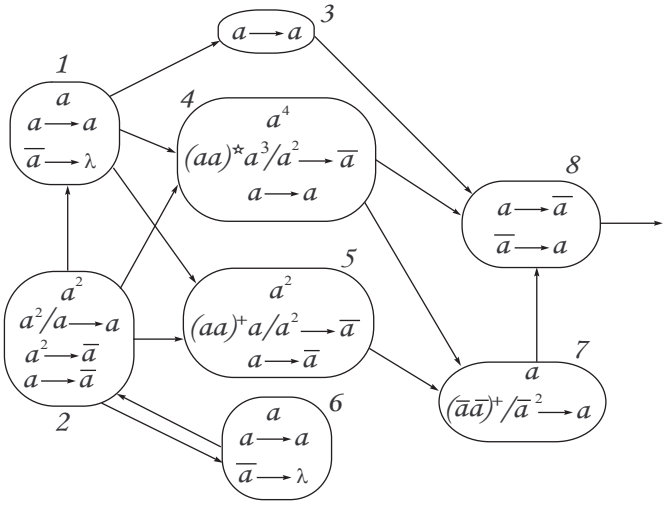


Figure 3.4: An SN PA system for the context sensitive language $\{0^n1^n0^n/n \geq 1\}$

In $n \geq 0$ steps when neuron σ_2 uses the first rule $a^2/a \to a$, the neurons σ_4 and σ_5 accumulates $2n+4$ and $2n+2$ spikes respectively. Neuron σ_1 sends a spikes to neuron σ_3, which in turn sends a spike to output neuron, which uses its first rule and sends an anti-spike(0) to environment. So the output is available from the third step onwards. At any step $n \geq 1$, when the neuron σ_2 uses the rule $a^2 \to \overline{a}$, the spike from neuron σ_1 and anti-spike from σ_2 will annihilate each other in neurons σ_4 and σ_5, left them with the same number of spikes as that of the previous step. Neuron σ_2 receives a spike from neuron σ_6 where as neuron σ_6 receives an anti-spike from neuron σ_2. In the same step spike from neuron σ_1 is also sent to neuron σ_3. In the

$n+1$ step neurons σ_1 and σ_6 forget their anti-spikes received from σ_2. Neuron σ_3 sends a spike to neuron σ_8. Neuron σ_2 uses its third rule $a \to \overline{a}$ by sending an anti-spike to neurons σ_4 and σ_5. Neurons σ_4 and σ_5 are left with $2n+3$ and $2n+1$ spikes respectively.The number of spikes in neurons σ_4 and σ_5 become odd, so they start firing using their first rules. In the $n+2$ step, the neuron σ_8 outputs a spike (that means total of $n+1$ spikes). Neuron σ_4 starts firing using the rule $(aa)^*a^3/a^2 \to \overline{a}$ repeatedly used until it is left with one spike, sending a total of $n+1$ anti-spikes to neurons σ_7 and σ_8. In σ_8, the anti-spikes are converted into spikes and sent to environment; At the same time σ_7 also receives anti-spikes from σ_5, accumulating a total of $2n+1$ anti-spikes (one spike initially present in it annihilates with one anti-spike). The last spike in neuron σ_4 is used by the second rule $a \to a$, and a spike is sent to output neuron and neuron σ_7. So the first anti-spike after $n+1$ comes from σ_4. As the neuron σ_7 is already having a spike, the total number of spikes become even $(2n)$. Then in the next n steps σ_7 fires by sending a spike to neuron σ_8, where it is converted into anti-spike and sent to environment. So the language generated becomes $\{0^n1^n0^n \mid n \geq 1\}$. □

3.2.5 A Characterization of Recursively Enumerable Languages

A characterization of recursively enumerable (RE) languages is possible in terms of languages generated by SN PA systems. Here we use the notion of a deterministic register machine. Such a device is a construct $M = (m, H, l_0, l_h, I)$, where m is the number of registers, H is the set of instruction labels, l_0 is the start label, l_h is the halt label (assigned to instruction HALT), and I is the set of instructions labelled in a one-to-one manner by the labels from H.

Theorem 3.5. *For every alphabet $V = \{a_1, a_2, \ldots, a_s\}$ there is a morphism $h : V^* \to B^*$ such that for each language $L \subseteq V^*$, $L \in RE$, there is an SN PA system Π such that $L = h^{-1}(L(\Pi))$.*

Proof. We follow here the same idea as in the proof of Theorem 9 from [18] adapted to the case of anti-spikes.
The morphism is defined as follows:
$h(a_i) = 0^i1$, for $i = 1, 2, \ldots, s$,

For a string $x \in V^*$, let us denote by $val_s(x)$, the value in base $s+1$ of x.(We use base $s+1$ in order to consider the symbols of $a_1, a_2, \ldots, a_s$ as digits $1, 2, \ldots s$, thus avoiding the digit 0 in the left hand of the string). We extend this notation in the natural way to the set of strings. Now consider a language $L \subseteq V^*$. Obviously $L \in RE$ iff $val_s(L)$ is recursively enumerable set of numbers. In turn, a set of numbers is recursively enumerable if and only if it can be accepted by a deterministic register machine [74]. Let M be such a register machine that is $N(M) = val_s(L)$.

We construct an SN PA system Π performing the following operations (σ_{c0} and σ_{c1} are two distinguished neurons of Π, which are empty in the initial configuration):

i. Output i anti-spikes, for some $1 \leq i \leq s$, and at the same time introduce the number i in neuron σ_{c0} ; in the construction below, a number n is represented in a neuron by storing there $2n$ spikes, hence the previous task means introducing $2i$ spikes in neuron σ_{c0}.

ii. When this operation is finished, output a spike hence up to now we have produced a string 0^i1.

iii. Multiply the number stored in neuron σ_{c1} (initially, we have here number 0) by $s+1$, then add the number from neuron σ_{c0} ; specifically, if neuron σ_{c0} holds $2i$ spikes and neuron σ_{c1} holds $2n$ spikes, $n \geq 0$; then we end this step with $2(n(s+1)+i)$ spikes in neuron σ_{c1} and no spike in neuron σ_{c0} : In the meantime, the system outputs no spike/anti-spike.

iv. Repeat from step 1, or, non-deterministically, stop the increase of spikes from neuron σ_{c1} and pass to the next step.

v. After the last increase of the number of spikes from neuron σ_{c1} we have got $val_s(x)$ for a string $x \in V^+$ such that the string produced by the system up to now is of the form $0^{i_1}1\lambda^{j_1}0^{i_2}1\lambda^{j_2}\ldots 0^{i_m}1\lambda^{j_m}$, for $1 \leq i_l \leq s$ and $j_l \geq 1$, for all $1 \leq l \leq m$. λ is a symbol for no output, which is ignored. i.e., $h(x) = 0^{i_1}10^{i_2}1\ldots 0^{i_m}1$. We now start to simulate the work of the register machine M in recognizing the number $val_s(x)$. During this process, we output no spike, but the computation halts if (and only if) the machine M halts, i.e., when it accepts the input number, which means that $x \in L$.

From the previous description of the work of Π, it is clear that the computation halts after producing a string of the form $y = 0^{i_1}1\lambda^{j_1}0^{i_2}1\lambda^{j_2}\ldots 0^{i_m}1\lambda^{j_m}\lambda^k$ as above, if and only if $x \in L$. Moreover, it is obvious that $x = h^{-1}(y)$: we have $h^{-1}(y) = a_{i_1}\ldots a_{i_m}$.

Now, it remains to construct the system Π. Instead of constructing it in all details, we rely on the fact that a register machine can be simulated by an SN PA system, as already shown in [79] for the sake of completeness and because of some minor changes in the construction, we below recall the details of this simulation. Then, we also suppose that the multiplication by $s+1$ of the contents of neuron σ_{c1} followed by adding a number between 1 and s is done by a register machine (with the numbers stored in neurons σ_{c0}, σ_{c1} introduced in two specified registers); we denote this machine by M_0. Thus, in our construction, also for this operation we can rely on the general way of simulating a register machine by an SN PA system. All other modules of the construction (introducing a number of spikes in neuron σ_{c0}, sending out spikes, choosing non-deterministically to end the string to generate and switching to the checking phase, etc.) are explicitly presented below.

A delicate problem which appears here is the fact that the simulations of both machines M_0 and M have to use the same neuron σ_{c1}, but the correct work of the system (the fact that the instructions of M_0 are not mixed with those of M) will be explained below. The overall appearance of Π is given in Figure 3.5, where M_0 indicates the subsystem corresponding to the simulation of the register machine

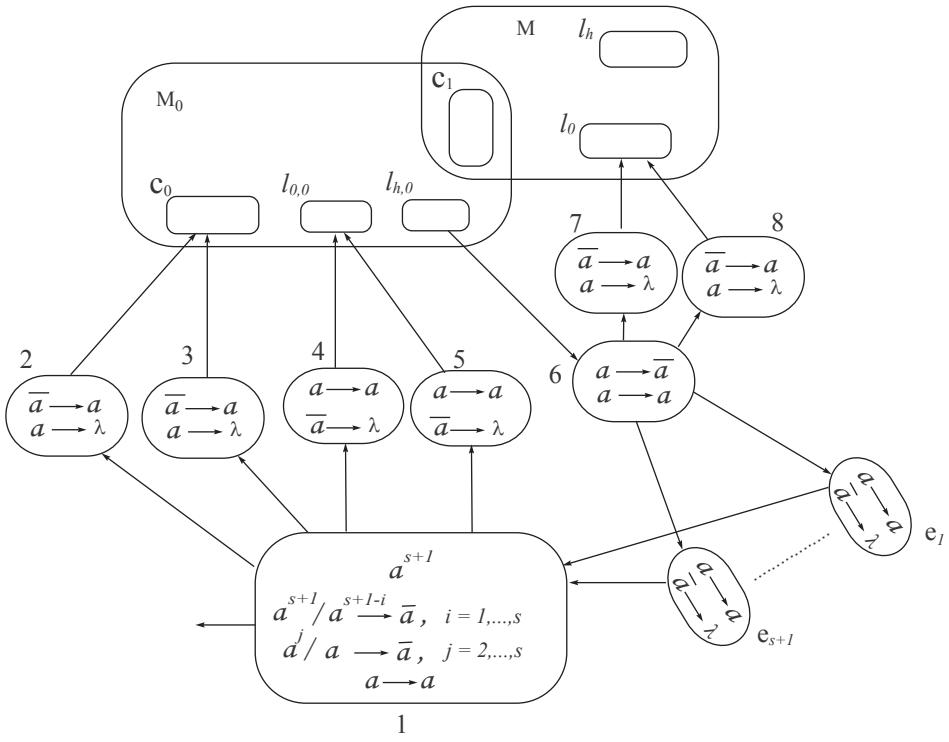


Figure 3.5: The structure of the SN PA system from the proof of Theorem 3.5

$M_0 = (m_0, H_0, l_{0,0}, l_{h,0}, I_0)$ and M indicates the subsystem which simulates the register machine $M = (m, H, l_0, l_h, I)$. Of course, we assume $H_0 \cap H = \emptyset$.
We start with $s+1$ spikes in neuron σ_1 and fires by using some rule $a^{s+1}/a^{s+1-i} \to \overline{a}$; $1 \leq i \leq s$, then in next $i-1$ steps, it uses its second rule producing a total of number i anti-spikes and the last spike is used by the third rule producing spike, hence the first letter a_i of the generated string. In each step, when neuron σ_1 is producing an anti-spike, 2 spikes are sent to the neuron σ_{c0} through the neurons σ_2 and σ_3, accumulating a total of $2i$ spikes and the step when the output neuron produces a spike, it is ignored by neurons σ_2 and σ_3 and two spikes are sent to neuron $l_{0,0}$; thus triggering the start of a computation in M_0.

The subsystem corresponding to the register machine M_0 starts to work, multiplying the value of $c1$ with $s+1$ and adding i. When this process halts, neuron $l_{h,0}$ is activated (this neuron will get two spikes in the end of the computation and will

spike), and in this way one spike is sent to neuron σ_6 : This is the neuron which non-deterministically chooses whether the string should be continued or we pass to the second phase of the computation, checking whether the produced string is in $L(M)$. In the first case, neuron σ_6 uses the rule $a \rightarrow a$; which makes neurons $e_1, \ldots, e_{s+1}$ spike; these neurons send $s+1$ spikes to neuron σ_1, like in the beginning of the computation. In the latter case, neuron σ_6 uses the rule $a \rightarrow \overline{a}$; which in turn activates the neurons σ_7 and σ_8, they activate l_0 by sending two spikes to it, thus starting the simulation of the register machine M. The computation of Π stops if and only if $val_s(x)$ is accepted by M. In order to complete the proof we need to show how

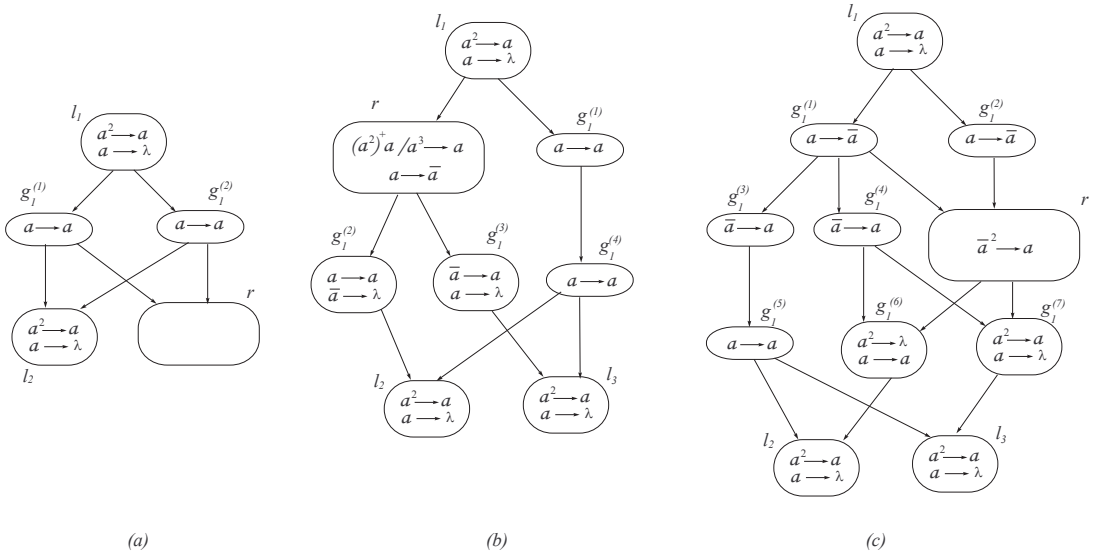


Figure 3.6: (*a*) Module ADD (simulating $l_i : (ADD(r), l_j)$) for M and M_0, Module SUB (simulating $l_i : (SUB(r), l_j; l_k)$) (*b*) for machine M and (*c*) for machine M_0

the two register machines are simulated, using the common neuron σ_{c1} but without mixing the computations. To this aim, we consider the modules ADD and SUB from Figure 3.6 Neurons are associated with each label of the machine (they fire if they have two spikes inside) and with each register (with $2n$ spikes representing the number n from the register), there also are additional neurons with labels ${g_1}^i$, $i \geq 1$ it is important to note that all these additional neurons have distinct labels.

The simulation of $(ADD(r), l_j)$ (add 1 to register r and then go to the instruction with label l_j) instruction is easy, we just add two spikes to the respective neuron; no rule is needed in the neuron Figure 3.6(a). The $(SUB(r), l_j, l_k)$ (if register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k) instructions of machines M and M_0 are simulated by modules as in Figure 3.6(b) and Figure 3.6(c), respectively. Note that the rules for M fire for a content of the neuron σ_r described by the regular expression $(a^2)^*a$ and the rules for M_0 fire for a content of the neuron σ_r described by the regular expression $(\overline{a})^2$. This ensures the fact that the rules of M_0 are not used instead of those of M or vice versa. With these explanations, it can be checked that the system Π works as requested. □

The previous theorem gives a characterization of recursively enumerable languages, because the family RE is closed under direct and inverse morphisms.

3.3 SN PA Systems as Transducers

This section explores the capability of SN PA systems as transducers. First we simulate the Boolean circuits. We show the construction of Boolean circuits using AND, OR and NOT involves the synchronizing module for yielding correct output. We provide an alternative way using universal gates - NAND and NOR which eliminates the use of synchronizing module. We also use these systems to simulate finite automata with output and binary arithmetic operations. Finally we solved the satisfiability problem using SN PA systems.

3.3.1 Simulating Boolean Gates

The Boolean values 0 and 1 are encoded in the SN PA system by anti-spike and spike respectively. The output of the system is 0(hence false) if the output neuron sends an anti-spike and output is 1(true) if a spike is sent to the environment. We want to emphasize that no rule of the form $\overline{a}^c \to \overline{a}$ is used.

Lemma 3.1. *Boolean AND and OR gates can be simulated by SN PA systems with three neurons in two steps.*

Proof. We construct an SN PA system with three neurons as in Figure 3.7. The SN PA system has two input neurons to take the input values and one output neuron to produce the output. A spike/anti-spike is introduced in each input neuron corresponding to input 1/0.

If we introduce an anti-spike (0) into each of the input neurons, the anti-spike becomes a spike and sent to the output neuron in the next stage. So the output neuron gets two spikes from the input neurons and it already has a spike, accumulating a total of three spikes and fires using a rule $a^3 \to \overline{a}$ sending an anti-spike (0) to the environment. But if we introduce a spike into each of the input neurons, the output neuron gets two anti-spikes and gets annihilated with a spike already present in it, remains with an anti-spike and fires using a rule $\overline{a} \to a$ producing a spike.

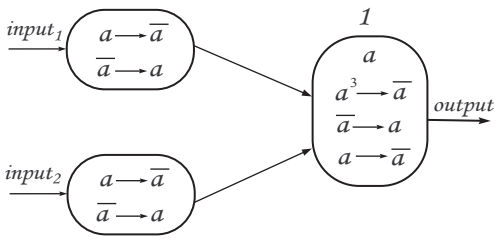


Figure 3.7: An SN PA system simulating AND gate

In the third case, if a spike is introduced into one of the input neurons and an anti-spike into another, then they get annihilated after reaching the output neuron. So the output neuron has its one spike and fires using the rule $a \rightarrow \overline{a}$ sending an anti-spike to the environment. We can observe that it is simulating the AND gate correctly.

If we replace the rule $a \rightarrow \overline{a}$ with $a \rightarrow a$ in the output neuron of the above system, we obtain the SN PA system for an OR gate. □

Lemma 3.2. *The Boolean NOT gate can be simulated by an SN PA system with two neurons in two steps.*

Proof. The SN P system with anti-spikes simulating the NOT gate is depicted in Figure 3.8. For synchronisation with OR and AND gates we added an output neuron so that output is produced after two steps. (Otherwise, the simulation is very simple, we can implement the gate with only one neuron in one step.)
If an anti-spike is introduced, the output neuron will have three spikes in the next

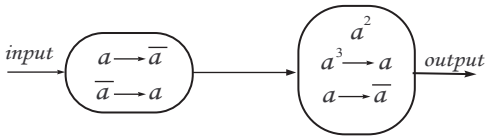


Figure 3.8: An SN PA system simulating NOT gate

step and fires using the rule $a^3 \rightarrow a$, sending a spike. If a spike is introduced, it gets complemented in the input neuron and annihilates with a spike in the output neuron in the next step. So the output neuron has only one spike and produces an anti-spike using the rule $a \rightarrow \overline{a}$. Thus the NOT gate complements the input. □

Lemma 3.3. *Boolean NAND and NOR gates can be simulated by SN PA systems with three neurons in two steps.*

Proof. We construct SN PA system with seven neurons as in Figure 3.9. The SN PA system has two input neurons to take the input values and one output neuron to produce output. A spike/anti-spike is introduced in each input neuron corresponding to input 1/0.

If we introduce an anti-spike (0) into each of the input neurons, the anti-spike becomes a spike and sent to the output neuron in the next stage. So the output neuron gets two spikes from the input neurons and it already has three spikes, accumulating a total of five spikes and fires using a rule $a^5 \rightarrow a$ sending a spike (1) to the environment. But if we introduce a spike (1) into each of the input neurons, the output neuron gets two anti-spikes and gets annihilated with two spikes already present in it, remains with a spike and fires using a rule $a \rightarrow \overline{a}$ producing an anti-spike (0). In the third case, if a spike is introduced into one of the input neurons

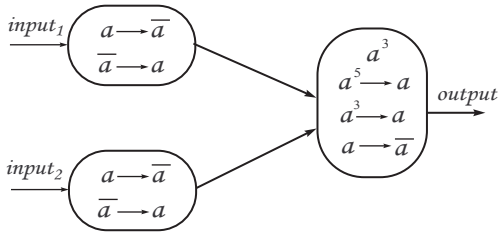


Figure 3.9: An SN PA system simulating 2-input NAND gate

and an anti-spike into another, then they get annihilated after reaching the output neuron. So the output neuron has its three spikes and fires using the rule $a^3 \rightarrow a$ sending a spike to the environment. We can observe that it is simulating the NAND gate in a correct way.

If we replace the rule $a^3 \rightarrow a$ with $a^3 \rightarrow \overline{a}$ in the output neuron of the above system, we obtain the SN PA system for the NOR gate.

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. The NAND gate represents the complement of the AND operation and the NOR gate represents

the complement of the OR operation. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

Similar to the 2-input NAND gate, we can construct n-input NAND gate. The out-

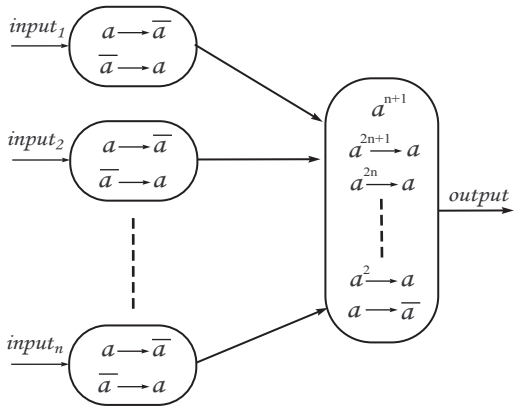


Figure 3.10: An SN PA system simulating n-input NAND gate

put of the gate is false (0) only if all the inputs are true (1) and is true if any of the inputs is false. The SN PA system for n-input NAND gate is shown in Figure 3.10. The maximum number of anti-spikes received by the output neuron is n (if all inputs are spikes corresponding to true) and they get annihilated with n spikes in the output neuron and is left with a spike and fires using the rule $a \rightarrow \overline{a}$ producing an anti-spike. In all other cases it produces a spikes. Thus simulating the n-input NAND gate correctly. □

3.3.2 Simulating Boolean Circuits

Here, we present the way to simulate any Boolean circuit using the AND, OR and NOT gates constructed in the previous section. But there is a need to construct synchronising module to ensure the synchronization among the gates.

We consider the function $f : \{0,1\}^4 \to \{0,1\}$ given by the formula $f(x_1, x_2, x_3, x_4) = \neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

We use the SN P systems with anti-spikes for AND, OR and NOT gates. Let them be Π_{AND}, Π_{OR} and Π_{NOT}. The Boolean circuit corresponding to the above formula as well as the spiking system assigned to it are depicted in Figure 3.11.

In order for the system that simulates the circuit to output the correct result it is necessary for each sub-system (that simulates the gates AND, OR and NOT) to receive the input from the above gate(s) at the same time. To this aim, we have to add a synchronizing SN P system Π_{SYN} as in Figure 3.12. Generalizing the previous observations the following result holds:

Theorem 3.6. *Every Boolean circuit, whose underlying graph structure is a rooted tree, can be simulated by an SN PA system constructed from SN PA systems of types AND, OR and NOT by reproducing in the architecture of the SN PA system, the structure of the tree associated with the circuit.*

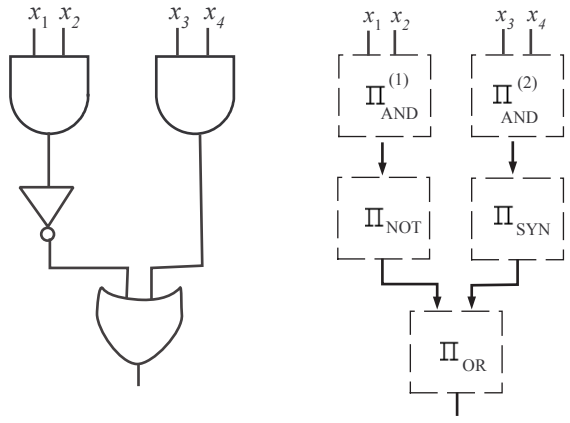


Figure 3.11: Boolean circuit and the corresponding SN PA system for $\neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Simulate any Boolean circuit using NAND or NOR gates eliminates the synchronizing SN PA system. We know that any Boolean function can be represented in sum-of-product (SOP) and product-of-sum forms (POS). SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only

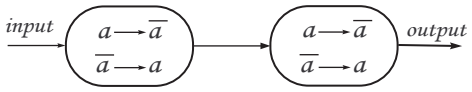


Figure 3.12: Synchronizing SN P system with anti-spikes

NOR gates. In either case, implementation requires two levels. The first level is for each term and second level for product or sum of the terms.

Again consider the Boolean function $\neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. It is written in SOP from as $\neg x_1 \vee \neg x_2 \vee (x_3 \wedge x_4)$.

We use the SN P systems with anti-spikes for 2-input and 3-input NAND gates. Let Π_{NAND} is an SN PA systems for NAND gate. The Boolean circuit corresponding to the above formula as well as the spiking system assigned to it are depicted in Figure 3.13.

Note that in Figure 3.13, $\Pi^{(1)}_{NAND}, \Pi^{(2)}_{NAND}, \Pi^{(3)}_{NAND}$ are SN PA systems for 2-input

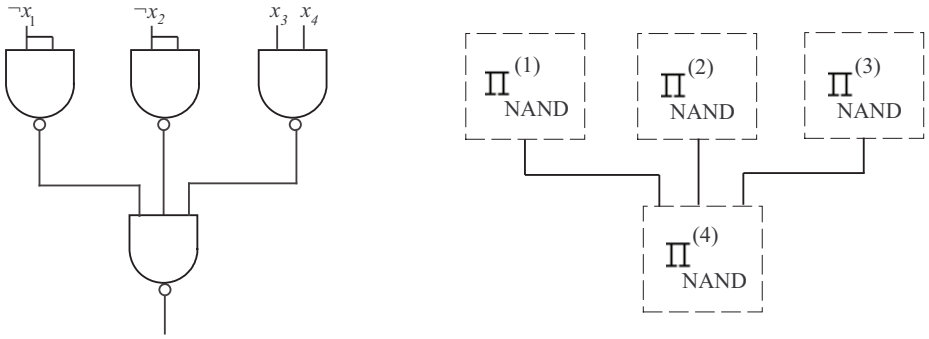


Figure 3.13: Boolean circuit using NAND gates and the corresponding SN PA system for $\neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

NAND gates and $\Pi^{(4)}_{NAND}$ is the SN P system for 3-input NAND gate. Having the overall image of the functioning of the system, let us give some more details on the simulation of the above formula. For that we construct the SN P system with anti-spikes.

Π_C=($\Pi^{(1)}_{NAND}, \Pi^{(2)}_{NAND}, \Pi^{(3)}_{NAND}, \Pi^{(4)}_{NAND}$) formed by the sub-SN PA systems for each gate and we obtain the unique result as follows:

1. For every gate of the circuit with inputs from the input gates we have a SN PA system to simulate it. The input is given to the input neurons of each gate;

2. For each gate which has at least one input coming as an output of a previous gate, we construct an SN PA system to simulate it by adding a synapse from the output neuron of the gate from which the signal (spike) comes to the input neuron of the system that simulates the new gate.

For the above formula and the circuit depicted in Figure 3.13 we will have:

$\Pi^{(1)}_{NAND}$ for the first NAND operation $\neg(\neg x_1 \wedge \neg x_1) = x_1$ with each input as $\neg x_1$. (for $\neg x_1$ as input, an anti-spike is introduced in each input neuron of $\Pi^{(1)}_{NAND}$).
$\Pi^{(2)}_{NAND}$ for the second NAND operation $\neg(\neg x_2 \wedge \neg x_2) = x_2$) with each input as $\neg x_2$.

$\Pi^{(3)}_{NAND}$ for the third NAND operation $\neg(x_3 \wedge x_4)$ with inputs as x_3 and x_4. These three SN PA systems $\Pi^{(1)}_{NAND}$, $\Pi^{(2)}_{NAND}$ and $\Pi^{(3)}_{NAND}$ act in parallel producing the output at the same time. The outputs enter the 3-input NAND gate $\Pi^{(4)}_{NAND}$ at the same time which eliminates the use of synchronising module.

$\Pi^{(4)}_{NAND}$ computes NAND operation on x_1, x_2 and $\neg(x_3 \wedge x_4)$ outputting $\neg x_1 \vee \neg x_2 \vee (x_3 \wedge x_4)$ to the environment.
Generalizing the previous observations the following result holds:

Theorem 3.7. *Every Boolean circuit can be simulated by an SN PA system and is constructed from SN PA systems of type NAND or NOR, by reproducing the structure associated with the circuit.*

3.3.3 Simulating Finite State Transducers

SN PA systems can simulate in a direct manner several types of computing devices based on finite state transitions. Let $S = (Q, \Sigma, \Delta, \delta, \mu, q_1, F)$ be a deterministic finite state transducer with binary input and output, where $\Sigma = \Delta = \{0, 1\}$, $Q = \{q_1, \ldots, q_n\}$, q_1 is the initial state, δ is the transition function that maps $Q \times \Sigma \to Q$ and μ is the output function from $Q \times \Sigma \to \Delta$.
We demonstrate that S can be simulated by an SN PA system.
Consider the following SN PA system:
Π_S=($\{a, \overline{a}\}$, σ_1, $\sigma_2, \ldots, \sigma_{3n+1}$, ***syn*** , $3n+1$, $3n+1$), with
σ_i = (a, $\{a \to a, a \to \overline{a}\}$), $i = 1, 2, \ldots, 3n$,
σ_{3n+1} = ($a^{3(n+1)}$, $\{a^{3(n+i)+1}/a^{3(n+i-j)+1} \to b' \mid \delta(q_i, 1) = (q_j, b)\} \cup$
$\{a^{3(n+i)-1}/a^{3(n+i-j)-1} \to b' \mid \delta(q_i, 0) = (q_j, b)\}$) where $b \in \{0, 1\}$ and $b' = a$ if $b = 1$ and $b' = \overline{a}$ if $b = 0$,
syn is the set of pairs $(i, 3n+i), (3n+i, i)$ with $1 \leq i \leq 3n$.

The system is given in a pictorial way in Figure 3.14. Note that n is the number of states, and that for each $1 \leq i \leq n$, q_i in Q is represented by $a^{3(n+i)}$. The number of spikes $a^{3(n+i)}$ in neuron σ_{3n+1} is referred to (or identified) as a state of Π_S. The manner of constructing Π_S is a modification of the one presented using extended SN P systems in [45].

This system works as follows. Initially, the neuron σ_{3n+1} contains $3(n+1)$ spikes which corresponds to the initial state q_1. Suppose that, in any step, neuron σ_{3n+1} contains $a^{3(n+i)}$(representing state q_i) and is ready to receive input a or $\overline{a}$ (representing 1 or 0 respectively) from environment. Depending on whether the input is a spike or anti-spike, neuron σ_{3n+1} can fire and emit a spike(if $b' = 1$) or anti-spike(if $b' = 0$) to environment by consuming $3(n+i-j)+1$ or $3(n+i-j)-1$ spikes leaving $3j$ spikes. It receives $3n$ spikes from neurons 1 to $3n$ accumulating a total of $3(n+j)$ spikes (representing q_j). Hence, one state transition $\delta(q_i, b) = (q_j, b')$ is

simulated. This action is repeatedly performed in a number steps equal to the input length. When the system stops receiving the input, the neuron σ_{3n+1} will have a number of spikes which is a multiples of 3, hence the system halts. Thus, (with one step delay) for a given input $w = b_{i_1} b_{i_2} \dots b_{i_r}$ in $\{0,1\}^*$, the SN PA system Π_S produces an output $y = \mu(q_{i_1}, b_{i_1})\mu(q_{i_2}, b_{i_2}) \dots \mu(q_{i_r}, b_{i_r})$ in $\{0,1\}^*$, where the sequence of states: $z = q_{i_1} q_{i_2} \dots q_{i_r}$ such that $\delta(q_{i_j}, b_{i_j}) = q_{i_{j+1}}$ for $j = 1, 2, \dots, r-1$ and $q_{i_1} = q_1$. We denote the output by $y = \Pi_S(w)$ and the sequence of states by $z = \Pi_{Sq}(w)$. A transducer S defines a function $w \rightarrow S(w)$, hence simulating S means that if $y = S(w)$, then $y = \Pi_S(w)$. Then it holds that y is generated by S (i.e., $\delta(q_1, w) \in F$) iff $z = \Pi_{Sq}(w)$ ends up with a final state in F (i.e., q_{i_r} is in F). We now define the language generated by Π_S as

$\mathcal{N}(\Pi_S) = \{y \in \{0,1\}^* \mid w \in \{0,1\}^*, y = \Pi_S(w)$ and $\Pi_{Sq}(w)$ is in $Q^*F\}$

Thus, the following theorem holds:

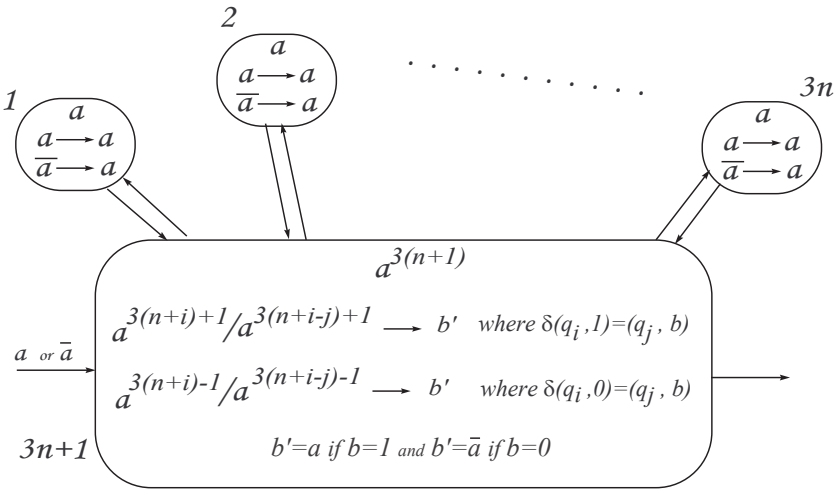


Figure 3.14: An SN PA system simulating a transducer

Theorem 3.8. *Any finite state transducer S can be simulated by some SN PA system* Π_S.

3.3.4 Arithmetic Operations using SN PA System

In this section we consider SN P system with anti-spikes as simple arithmetic device that can perform the arithmetic operations like 2's complement, addition and subtraction with input and output in binary form. The binary sequence of 0 and 1 are encoded as anti-spike and spike respectively and in each time step input is provided bit-by-bit starting from least significant bit. The negative numbers are represented in two's complement form. The advantage of using SN P systems with anti-spikes is that they can encode the 0 and 1 as anti-spike and spike in a very natural way and thus providing a way to represent negative numbers also.

2's Complement

The 2's complement is used to represent a negative of a binary number. It also gives us a straightforward way to add and subtract positive and negative binary numbers. A simple way to find the 2's complement of a number is to start from the least significant bit keeping every 0 as it is until you reach the first 1 and then complement all the rest of the bits after the first 1.

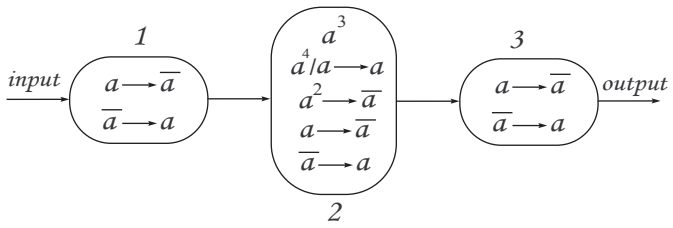


Figure 3.15: An SN PA system computing 2's complement

Theorem 3.9. *2's complement of a binary number can be calculated using an SN P systems with anti-spikes using three neurons.*

Proof. The SN PA system that performs the 2's complement is shown in Figure 3.15. Neuron σ_1 is the input neuron. Neuron σ_3 is the output neuron, which sends output to environment. The input neuron has two rules to complement the input by changing a spike into anti-spike and anti-spike into spike and send it to its neighbouring neuron σ_2. The neuron σ_2 initially has 3 spikes and as long as it receives a spike (actual input to the input neuron is 0), it uses the first rule $a^4/a \rightarrow a$ by send a spike to the output neuron where it is complemented into anti-spike, which is same as the input. But if the second neuron receives anti-spike(that means we got the first 1), it will be left with two spikes because of the annihilation rule that is implicitly present in each neuron and uses the rule $a^2 \rightarrow \overline{a}$ and sends an anti-spike to the output neuron where it is complemented as spike and sent to the environment(that is first 1 is unchanged). After firing the rule, the neuron σ_2 has no spikes/anti-spikes and then simply complements the input it receives by using the third and fourth rule and sends it to the output neuron where it is again complemented and sent to environment. That means after the first one, the output will be the complement of input. We can easily observe that the system correctly calculates the 2's complement and emits its first output bit at $t = 4$ as there is one intermediate neuron. □

As an example, let us consider a binary number 01100 (12 in decimal). The way the SN PA system computes the 2's complement is represented in Table 3.1. It reports the number of spikes/anti-spikes present in each neuron and output produced by the output neuron to the environment in the output column.

Addition and Subtraction

The SN PA system performing the addition is shown in Figure 3.16. The negative numbers are represented in 2's complement form using the system SN PA system given in the previous section and then fed as input.

Time	Input	Neuron σ_2	Neuron σ_3	Output
t=0	-	a^3	-	-
t=1	$\overline{a}(0)$	a^3	-	-
t=2	$\overline{a}(0)$	a^4	-	-
t=3	$a(1)$	a^4	a	-
t=4	$a(1)$	a^2	a	$\overline{a}(0)$
t=5	$\overline{a}(0)$	$\overline{a}$	$\overline{a}$	$\overline{a}(0)$
t=6	-	a	a	$a(1)$
t=7	-	-	$\overline{a}$	$\overline{a}(0)$
t=8	-	-	-	$a(1)$

Table 3.1: Number of spikes/anti-spikes present in each neuron of an SN PA system during the computation of 2's complement of 01100.

Theorem 3.10. *Addition of two binary numbers can be performed using SN P systems with anti-spikes.*

Proof. The system has two input neurons, the first number is provided through first input neuron and the second one through the other input neuron. First input neuron is connected to neurons σ_1 and σ_2 and second input neuron is connected to neurons σ_3 and σ_4. The presence of a spike in the output neuron indicates a carry of the previous addition. Each input neuron has two rules to complement the input and send the output to its neighbouring two neurons. here we are having 3 cases:

1. If both the inputs are 1(spike), then in each input neuron uses the second rules and sends an anti-spike two of its neighbouring neurons where the anti-spikes are converted spikes. So the output neuron σ_5 receives four spikes, one from each of the four neurons of the previous stage. If the output neuron is already having a spike(carry), then the number of spikes become 5 and fires using a rule $a^5/a^4 \rightarrow a$ otherwise it has four spikes and fires using the rule $a^4/a^3 \rightarrow \overline{a}$

leaving one spike in the output neuron in either case. The presence of a spike in the output neuron indicates a carry. This encodes the two operations 1+1=0 with carry 1 and (1)+1+1=1 with carry 1.

2. If one of the input bit is zero, then the input neuron receiving an anti-spike sends a spike to each of it's neighbouring neurons. For example if the first input is 0 and the second input is 1 then first input neuron sends a spike two each of neighbouring neurons σ_1 and σ_2. In the neuron σ_1, the spike remain the same and where as in σ_2 it is forgotten, so the number of spikes sent to the output neuron is 1, whereas the neighbouring neurons of second input neuron send two spikes to the output neuron. So three spikes are received if one of the input is zero. The output neuron has either three or four (in case carry) spikes and fires using $a^3 \rightarrow a$ or $a^4/a^3 \rightarrow \overline{a}$ respectively. These rules encode the two operations 0+1=1 and (1)+0+1=0 with carry 1 respectively.

3. If both the input neurons receive anti-spikes (0), then the output neuron receives two spikes and it will have either two or three (again in case of carry of the previous operation) spikes and fires using $a^2 \rightarrow \overline{a}$ or $a^3 \rightarrow a$. These two rules do not leave any carry encoding the operations 0+0=0 and (1)+0+0=1 respectively.

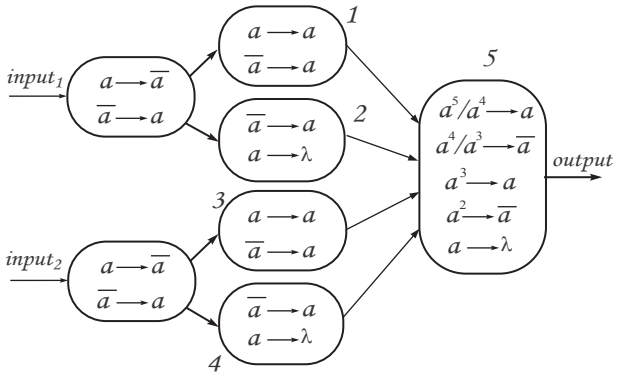


Figure 3.16: An SN PA System simulating addition operation

Time	Input 1	Input 2	σ_1	σ_2	σ_3	σ_4	σ_5	Output
t=1	$a(1)$	$a(1)$	-	-	-	-	-	-
t=2	$a(1)$	a (1)	$\overline{a}$	$\overline{a}$	$\overline{a}$	$\overline{a}$	-	-
t=3	$a(1)$	$\overline{a}(0)$	$\overline{a}$	$\overline{a}$	$\overline{a}$	$\overline{a}$	a^4	-
t=4	$\overline{a}(0)$	$a(1)$	$\overline{a}$	$\overline{a}$	a	a	a^5	$\overline{a}(0)$
t=5	-	-	a	a	$\overline{a}$	$\overline{a}$	a^4	$a(1)$
t=6	-	-	-	-	-	-	a^4	$\overline{a}(0)$
t=7	-	-	-	-	-	-	a	$\overline{a}(0)$
t=7	-	-	-	-	-	-	-	-

Table 3.2: Number of spikes/anti-spikes present in each neuron of the SN PA system during the addition of 0111 and 1011.

The last rule in the output neuron $a \to \lambda$ allows the last overflow bit to be ignored. The procedure confirms the correctness of the system for performing the addition of two numbers. □

As an example, let us consider the addition of 7 and -5. Number 7 is represented in binary form as 0111 and -5 is represented in 2's complement form as 1011. The two binary sequences will form the input for the SN PA system. The number of spikes present in each neuron in every step and the output produced by the system is depicted in Table 3.2.

Two's complement subtraction is the binary addition of the minuend to the 2's complement of the subtrahend (adding a negative number is the same as subtracting a positive one). That means $a - b$ becomes $a + (-b)$. The SN PA system for addition can be used to perform subtraction. The multiplication is viewed as repeated addition and division as repeated subtraction. This implies that SN P systems with anti-spikes can very well perform the binary operations in a natural way.

3.4 Solving SAT with SN PA Systems

An instance of SAT is a Boolean formula in CNF $\gamma = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, i.e., a conjunction of clauses C_j, $1 \leq j \leq m$. Each clause is a disjunction of literals, i.e., occurrences of x_i or $\neg x_i$, built on the finite set $X = \{x_1, x_2, \ldots, x_n\}$ of Boolean variables. In what follows, we will require that no repetitions of the same literal may occur in any clause; in this way, a clause can be seen as a subset of all possible literals. An assignment of the variables $x_1, x_2, \ldots, x_n$ is a mapping $s : X \to \{0,1\}^n$ that associates to each variable a truth value. The number of all possible assignments to the variables of X is 2^n. We say that Boolean formula γ is *satisfiable* if there exists an assignment of truth values to all the variables which occur in γ such that evaluation of γ gives 1 (true) as a result. The problem of SAT takes an arbitrary Boolean formula γ as input and asks if γ is satisfiable.

An SN PA system that solves the SAT problem in a non-deterministic uniform way is given in Figure 3.17. The system has one module for each clause. As the construction is uniform, we code each clause C_j, $1 \leq j \leq m$, of the given instance of SAT as follows: 1 indicates the case when x_i appears in C_j, 0 indicates the case when $\neg x_i$ appears in C_j and λ (empty) indicates the absence of x_i in the clause C_j. That means that a spike, an anti-spike or no input (λ) are to be introduced in the input neurons of the system from the second step onwards and the output neuron emits a spike, if the given instance of SAT has a solution, otherwise sends an anti-spike.

Actually, we consider m input neurons, one for each clause, and in each of them we introduce a sequence of n bits 1, 0 and λ (a spike, anti-spike or no input is sent inside in the steps corresponding to the occurrence of 1,0 and λ respectively), describing the situation of each variable $x_1, \ldots, x_n$ with respect to the corresponding clause.

For instance, for the formula

$\gamma = (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3)$

we have two input neurons, the first one receiving the spike train $0\lambda 10$, and the second one receiving the spike train $\lambda 10\lambda$. Note the important fact that introducing the input takes only n steps.
A module Y_j exists for each clause C_j, $1 \leq j \leq m$. Each module has a synapse going to the output neuron σ_{out}.

Neurons σ_c and σ_b are common to all modules; a synapse exists from σ_c to σ_b and from σ_b to all neurons σ_{d_j} of modules Y_j. Neuron σ_c provides a spike to neuron σ_b in each step for $n-1$ steps. The neuron σ_b non-deterministically produces a truth-assignment for the variables $x_1, \ldots, x_n$, using the choice between rules $a \to a$ and $a \to \overline{a}$. The spike needed for the truth assignment of x_1 is initially present in the neuron σ_b, while it gets the spike in each step for the next $n-1$ variables from the neuron σ_c.

An anti-spike coming out of σ_b is interpreted as the value *false* assigned to x_i, and a spike is interpreted as the value *true* assigned to x_i. Therefore, σ_{d_j} receives either an anti-spike or a spike from the neuron σ_b, and the spikes which codify the type of presence of x_i in clause C_j (no occurrence, negated, not negated).

In order to synchronize the checking performed in neurons σ_{d_j}, i.e., to bring here the truth assignment of variable x_i in the moment when the code of the presence of x_i in C_j arrives in this neuron, we use the neuron σ_c that supply spikes to neuron σ_b in each step for next $n-1$ steps. In each step beginning with the second step, all neurons σ_{d_j} receive both the truth assignment of x_i and the code of the way x_i is related with C_j. As one can see from the previous explanations, in each step $2, 3, \ldots, n+1$, neurons σ_{d_j}, $1 \leq j \leq m$, receive a number of spikes/anti-spikes as follows:
1 anti-spike if x_i = false and x_i does not appear in C_j;
1 spike if x_i= true and x_i does not appear in C_j;
no spikes/anti-spikes if x_i= false and x_i appears in C_j;
2 spikes if x_i= true and x_i appears in C_j;

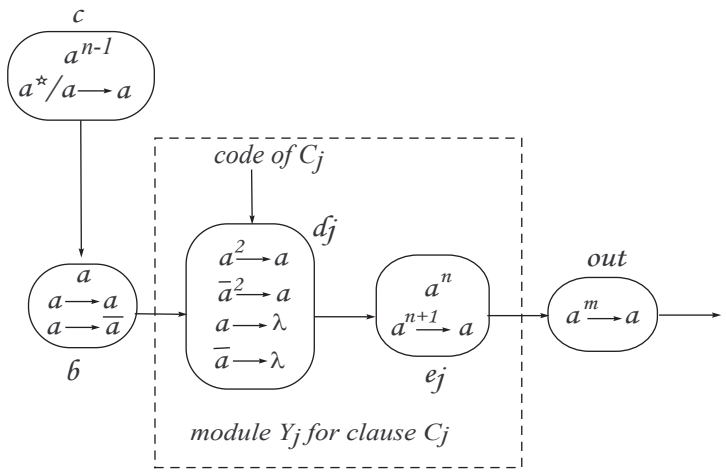


Figure 3.17: An SN PA system solving SAT

2 anti-spikes if x_i = false and $\neg x_i$ appears in C_j;
no spikes/anti-spikes if x_i= true and $\neg x_i$ appears in C_j

Thus, the rules of σ_{d_j} produce a spike only in the case when the clause C_j becomes true for the corresponding truth assignment of variable x_i. This spike reaches neurons σ_{e_j}. Each neuron σ_{e_j} has already n spikes and fires using the rule $a^{n+1} \to a$. The use of neuron e_j ensures the fact that σ_{out} receives at most one spike from each module Y_j, namely, only if clause C_j has been satisfied. All neurons σ_{e_j}, $1 \leq j \leq m$, are linked by a synapse to the output neuron σ_{out}. The neuron σ_{out} spikes (in step $n+3$) using a rule $a^m \to a$ only if the truth assignment produced non-deterministically by modules Y_j satisfies the formula γ.

It should be noted that the number of neurons of the system constructed above is $2m+3$, and that the computation takes a number of steps which is linear in n. Note that without anti-spikes [66] the solution requires double the number of steps and $3n^2+8m+5$ neurons.

3.5 Conclusion

In this chapter, we have investigated the power of SN PA systems with standard rules as language generators. We have proved characterizations of finite and regular languages over binary alphabet. We can extend the proofs to any alphabet by considering the morphisms. We have also proved a characterization of recursively enumerable languages. Here we ignored the no output steps.

We also examined the computational efficiency of SN PA systems used as transducers. We showed that the idea of encoding 1 as spike and 0 as anti-spike proves to be very efficient in simulating Boolean circuits, finite state transducers and solving NP-complete problems. We designed SN PA systems simulating the operations of different Boolean gates. We also designed SN P systems with anti-spikes to perform arithmetic operations like 2's complement, addition and subtraction. The advantage of using this variant of SN P system is that we can perform the operations on negative numbers also. The input to the systems is a binary sequence of spikes and anti-spikes which encodes the digits 1 and 0 respectively, of a binary number. The negative numbers are in 2's complement form. The outputs of the computations are also expelled to the environment in the same form. This motivates the modelling of CPU with SN P system with anti-spikes. We show that any instance of SAT in conjunctive normal form, with n variables and m clauses is solved in a nondeterministic way with the number of neurons polynomial in m.

Chapter 4

SN P Systems and Petri Nets

This chapter describes an approach to the modelling of spiking neural P systems using Petri nets. The approach is supported by simulating and analysing the obtained models through a Java based Petri net tool called PNetLab. The simulation and analysis results are demonstrated through examples. This enables us to employ Petri nets for analyzing the computation and the behavioural properties of SN P systems. In Section 4.2, we define vector rules and transitions of the SN P systems in order to relate them with the Petri nets. In Section 4.3, we introduce the class of Petri nets we are interested in and the execution modes to be considered for the simulation of SN P systems. General procedure to translate standard SN P systems into equivalent Petri net models is provided in Section 4.4. The algorithm is illustrated through series of examples in Section 4.5. The simulation and analysis results are presented for the SN P systems through PNetLab.

4.1 Introduction

An SN P system contains a set of neurons, each neuron can hold spikes in the form of occurrences of a symbol a; the spiking activity of neurons in the brain is abstracted to spiking and forgetting rules associated with conditions represented as regular expressions over $\{a\}$. SN P systems work in a locally sequential and globally parallel way. That is, in each neuron, at each step, if more than one rule is enabled, then only one of them can fire. But still, all neurons fire in parallel at the system level.

It is usually a complex task to predict or to guess how an SN P system will behave. Moreover, as there do not exist, up to now, implementations in laboratories (neither *in vitro* nor *in vivo* nor in any electronic media), it seems natural to look for software tools that can provide assistance to simulate the computations of SN P systems.

In [39], a tool for simulating standard and extended SN P system is introduced that yields only the transition diagram of a given system in a step-by-step mode and it lacks of step-by-step graphical simulation and analysis of these systems.

In [68], a P-Lingua based simulator for SN P systems with neuron division and budding is presented. P-Lingua is a programming language to define P systems [1, 22, 26], that comes together with a Java library providing several services; (e.g., parsers for input files and built-in simulators). The new version has extension of the previous syntax in order to define SN P systems with neuron division and budding and the library has been updated to handle P-Lingua input files defining SN P systems. A new built-in simulator has been added to the library in order to simulate computations of SN P systems. The SN P system variants considered in the thesis are not covered in P-Lingua.

In [15], SN P systems without delay are simulated using a highly parallel computing device such as a Graphical processing units (GPU) and the NVIDIA Compute Unified Device Architecture (CUDA) programming model. The simulator was

shown to model the working of an SN P system without delay using the system's matrix representation [99]. Simulating SN P systems in parallel devices such as GPUs need an extra hardware and software.

There are different interactions between classes of P systems and Petri nets that have been investigated so far [25, 55–60]. Different variants of P systems are translated into Petri nets to complement the functional characterisation of their behaviour. In [59, 60] Petri nets with localities are introduced to represent compartmentisation of membrane systems. Range arcs are introduced in Petri nets [57, 58] to model the dynamic structure of membranes, inhibitors and promoters in membranes . However, all these new variants of Petri nets typically lack the tools for building models, for executing, and for observing simulation experiments.

The translation of spiking neural P systems into models of Petri nets was first mentioned in [33], suggestion for research in Section 6. In this chapter, we propose to use Petri nets with guard function as a model for the semantics of parallel systems like SN P systems. Petri nets are widely used as a model of concurrency, which allows to represent the occurrence of independent events. They can be as well a model of parallelism [53], where the simultaneity of the events is more important, when we consider their step sequence semantics in which an execution is represented by a sequence of steps, each of them being the simultaneous occurrences of transitions. The step sequence semantics can very well represent the locally sequential and globally maximal firing semantics of standard SN P systems. A major strength of Petri nets is their support for analysis of many properties and problems associated with concurrent systems. Petri nets are thus suitable for specifying and verifying SN P systems.

In this chapter, we consider P/T nets with marking-dependent arc weights and guard functions for modelling SN P systems. The Petri net models obtained after translation are considered for simulation using PNetLab. PNetLab is a Java based Petri net tool which supports the parallel execution of transitions. It also allows to

write user defined guard functions in C/C++, which makes it possible to represent the regular expressions associated with spiking/forgetting rules. It also provides step-by-step watching system for collecting simulation reports. Our main goal in this chapter is to show that Petri nets are also suitable for the execution and to study the behavioural properties of the modelled SN P systems.

4.2 Standard SN P Systems

An SN P system $\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, syn, i_0)$ is represented as a directed graph where nodes correspond to the neurons having spiking and forgetting rules. The rules involve the spikes present in the neuron in the form of occurrences of a symbol a. The arcs indicate the synapses among the neurons. The spiking rules are of the form $E/a^r \rightarrow a; d$ and are used only if the neuron contains n spikes such that $a^n \in L(E)$ and $n \geq r$, where $L(E)$ is the language generated by the regular expression E. In this case r number of spikes are consumed and one spike is sent out. When neuron σ_i sends a spike, it is replicated in such a way that one spike is sent to all neurons σ_k such that $(i, k) \in syn$ after d steps, where syn is the set of directed arcs between the neurons. Between the moment when a neuron fires and the moment when it spikes, each neuron needs a time interval called delay d (or refractory period). During the delay, the neuron is closed and it neither fires nor receives any spikes from other neurons. The forgetting rules are of the form $a^s \rightarrow \lambda$ and are applied only if the neuron contains exactly s spikes. The rule simply removes s spikes. For all forgetting rules, a^s must not be the member of $L(E)$ for any firing rule $E/a^r \rightarrow a; d$ within the same neuron.

If $d = 0$, then sometimes it is omitted when writing the rule and are called non delayed rules. If all rules in system are non delayed (i.e. $d = 0$ in all rules) then the system is called SN P system without delay.

Definition 4.1 (**Configuration**). *The configuration of the system is described by both*

the contents of each neuron and its state, which can be expressed as the number of steps to wait until it becomes open (zero if the neuron is already open). Thus $\langle \alpha_1/d_1, \alpha_2/d_2, \ldots, \alpha_m/d_m \rangle$ *is a configuration where neuron* σ_i *contains* $\alpha_i \geq 0$ *spikes and it will open after* $d_i \geq 0$ *steps, for* $i = 1, 2, 3, \ldots, m$*. With this notation, the initial configuration of the system is described by* $C_0 = \langle n_1/0, n_2/0, n_3/0, \ldots, n_m/0 \rangle$.

The configuration of an SN P system without delay is represented by omitting the delay part as $\langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$. The SN P system is synchronized by means of a global clock and works in a locally sequential and globally maximal manner. That is, the working is sequential at the level of each neuron. In each neuron, at each step, if more than one rule is enabled by its current contents, then only one of them (chosen non-deterministically) can fire. But still, the system as a whole evolves in parallel and in a synchronising way, as in, at each step, all the neurons (that have an enabled rule) choose a rule and all of them fire at once.

Definition 4.2 (**Vector rule**)**.** *A vector rule of* Π *is a tuple* $\boldsymbol{v} \stackrel{df}{=} \langle \boldsymbol{1j}_1, \boldsymbol{2j}_2, \ldots, \boldsymbol{mj}_m \rangle$ *where, for each neuron* σ_i*,* $\boldsymbol{j}_i$ *is either* ***0*** *(when no rule is enabled in* σ_i*) or* ***s*** *(****s*** *stands for spiking of the neuron* σ_i *after being closed for* $d - 1$ *steps) or an enabled rule* $\boldsymbol{ij}_i$ *from* R_i*.*

If a vector rule $\boldsymbol{v}$ of Π is enabled at a configuration $C = \langle n_1/d_1, n_2/d_2, \ldots, n_m/d_m \rangle$, then $\boldsymbol{ij}_i$ for each neuron σ_i can be in any of the following forms:

1. ***i0***, if σ_i is closed and $d_i \geq 2$ or no rule is enabled.
2. ***is***, if $d_i = 1$.
3. ***ij***, if σ_i is open and $n_i \in \Psi(L(E))$ for any rule $\boldsymbol{ij} \in R_i$ with regular expression E.

If more than one rule is enabled, then the rule ***ij*** will be chosen non-deterministically. The number of spikes consumed and produced by the rule ***ij*** is denoted as *lhs(**ij**)* and *rhs(**ij**)* respectively. Note that *rhs(**is**)*=0 and *rhs(**i0**)*=0.

Definition 4.3 (Transition). *A vector rule* $\boldsymbol{v} = \langle \boldsymbol{1j}_1, \boldsymbol{2j}_2, \ldots, \boldsymbol{mj}_m \rangle$ *enabled at* $\mathcal{C} = \langle \alpha_1/d_1, \alpha_2/d_2, \ldots, \alpha_m/d_m \rangle$ *can evolve to* $\mathcal{C}' = \langle \alpha'_1/d'_1, \alpha'_2/d'_2, \ldots, \alpha'_m/d'_m \rangle$ *such that for every* σ_i *in* Π*:*

$$d'_i = \begin{cases} d & \text{if } \boldsymbol{ij}_i \text{ is of the form } E/a^r \to a; d \text{ with } d \geq 1 \\ d_i - 1 & \text{if } \boldsymbol{j}_i = \boldsymbol{0} \text{ and } d_i \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$n'_i = \begin{cases} n_i - lhs(\boldsymbol{ij}_i) + \sum_{(k,i) \in syn} rhs(\boldsymbol{kj}_k) & \text{if } \boldsymbol{ij}_i \text{ is of the form } E/a^r \to a; 0 \text{ or } a^s \to \lambda \\ n_i - lhs(\boldsymbol{ij}_i) & \text{if } \boldsymbol{ij}_i \text{ is of the form } E/a^r \to a; d \text{ with } d \geq 1 \\ n_i + \sum_{(k,i) \in syn} rhs(\boldsymbol{kj}_k) & \text{if } \boldsymbol{ij}_i = \boldsymbol{is} \\ n_i & \text{if } \boldsymbol{ij}_i = \boldsymbol{i0} \end{cases}$$

Using a vector rule $\boldsymbol{v}$, we pass from one configuration of the system to another configuration and such a step is called a transition. We denote this by $\mathcal{C} \overset{\boldsymbol{v}}{\Longrightarrow} \mathcal{C}'$. Note that the transition of $\mathcal{C}$ is non-deterministic in the sense that there may be different vector rules applicable to $\mathcal{C}$, as described above.

A *computation* of Π is a finite or infinite sequences of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called reachable. A computation halts if it reaches a configuration where no rule can be used.

Let $\gamma = \mathcal{C}_0 \overset{\boldsymbol{v}_1}{\Longrightarrow} \mathcal{C}_1 \overset{\boldsymbol{v}_2}{\Longrightarrow} \ldots \overset{\boldsymbol{v}_k}{\Longrightarrow} \mathcal{C}_k$ be an halting computation ($\mathcal{C}_0$ is the initial configuration, and $\mathcal{C}_{i-1} \overset{\boldsymbol{v}_i}{\Longrightarrow} \mathcal{C}_i$ is the ith transition of γ).
For each transition i of γ, we associate a symbol $bin(\boldsymbol{v}_i) = b_i \in \{0,1\}$ such that $b_i = 1$ if and only if $\boldsymbol{v}_i(i_0)$ is a non delayed spiking rule with $rhs(\boldsymbol{v}_i(i_0)) = 1$ or $\boldsymbol{v}_i(i_0) = \boldsymbol{is}$ (output neuron of the system Π sends a spike into the environment in transition i of γ), otherwise $b_i = 0$. Let us denote by $bin(\gamma)$ the string $b_1 b_2 \ldots b_k$ where $bin(\boldsymbol{v}_i) = b_i$ for $i = 1, 2, \ldots k$.

We denote by $COM(\Pi)$ the set of all halting computations of Π. Moreover, the language generated by Π is defined as $L(\Pi) = \{bin(\gamma) \mid \gamma \in COM(\Pi)\}$ [18].

4.3 Petri Nets with Guard

Now, we define a class of Petri nets suitable for the parallel execution of SN P systems.

Definition 4.4 (**P/T net with marking-dependent arc weights and guard functions or Petri net with guard**). *A Petri net with guard is a tuple* $\mathcal{NL} = (P, T, F, W, G, \mathcal{M}_0)$, *where*
$P = \{p_1, p_2, \ldots p_m\}$ *is a finite, non-empty set of places; A marking* $\mathcal{M} = \{\mathcal{M}(p_1), \mathcal{M}(p_2), \ldots, \mathcal{M}(p_m)\}$ *describes an assignment of tokens to each place. A marking-dependent expression is a function of the marking,* $f(\mathcal{M}) = f(\mathcal{M}(p_1), \mathcal{M}(p_2), \ldots, \mathcal{M}(p_m)) \in \mathbb{N}$.
$T = \{t_1, t_2, \ldots t_n\}$ *is a finite, non-empty set of transitions;*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs;*
$W : F \to f(\mathcal{M})$ *is a weight function; and*
for each $t \in T$, $G(t)$ *is a boolean expression called the guard of* t *which is again marking-dependent. It specifies an additional constraint and the transition* t *is enabled only if* $G(t)$ *holds true in the marking.*
$\mathcal{M}_0 : P \to \mathbb{N}$ *is the initial marking.*

The terms Petri nets and P/T nets with marking-dependent arc weights and guard functions are used synonymously in this chapter and further in the thesis. A marking $\mathcal{M}$ of $\mathcal{NL}$ is a function from the set of places of $\mathcal{NL}$ into the set of non negative integers $\{0, 1, 2, \ldots\}$. *Submarking* of a Petri net $\mathcal{NL}$ is the marking of some of the places of $\mathcal{NL}$.

Parallel activities can be easily expressed in terms of Petri nets using step semantics in which an execution is represented by a sequence of steps [13, 53]. Steps

in Petri nets are sets of transitions that fire independently and parallel at the same time. The change in the marking of the net when a step occurs is given by the sum of all the changes that occur for each transition. As in [53], if every transition occurs only once, like in elementary net systems, thus having single usage within the step, then we use the naming step. A multiset of transition, on the other hand, may contain more than one occurrence of a transition. In this case, a transition can fire several times in one step. Since such multisets of transitions can be seen as the sum of several single steps, they will shortly be called multi-steps. For translating standard SN P systems, we consider only steps.

Definition 4.5 (**Step occurrence rule**). *Let $\mathcal{NL} = (P, T, F, W, G, \mathcal{M}_0)$ be a Petri net. A set of transitions $U \in 2^T/\{\emptyset\}$, called step, is enabled to occur in a marking $\mathcal{M}$ of $\mathcal{NL}$ iff*
$\forall p \in P : \mathcal{M}(p) \geq \sum_{t \in U} W(p,t)$ *and* $\forall t \in U, G(t)$ *is true.*
If a step U is enabled to occur in a marking $\mathcal{M}$, then its occurrence leads to the new marking $\mathcal{M}'$ defined by
$\forall p \in P : \mathcal{M}'(p) = \mathcal{M}(p) + \sum_{t \in U}(W(t,p) - W(p,t))$

We write $\mathcal{M}[U\rangle\mathcal{M}'$ to denote that U is enabled to occur in $\mathcal{M}$ and that its occurrence leads to $\mathcal{M}'$. It is worth noting that if a step U is enabled at a marking, then so is any sub-step $U' \subseteq U$. A step U is a maximal step at a marking $\mathcal{M}$, if $\mathcal{M}[U\rangle$ and there is no transition t' such that $\mathcal{M}[U + t'\rangle$. A Petri net system $\mathcal{NL}$ with *maximal strategy* is such that for each markings $\mathcal{M}$ and $\mathcal{M}'$ if there is a step U such that $\mathcal{M}[U\rangle\mathcal{M}'$, then U is a maximal step and we write as $\mathcal{M}[U\rangle_m\mathcal{M}'$. In [13], it is proved that P/T nets with maximal strategy can perform the test for zero and so the computational power is extended up to the power of Turing machines. Ciardo in [20] proved that P/T nets with marking-dependent arc weights are also Turing equivalent.

The notions of steps and maximal steps are recursively generalized to step sequences and maximal step sequences:

Definition 4.6 (**Step Sequence**). *Let $\mathcal{NL} = (P, T, F, W, G, \mathcal{M}_0)$ be a Petri net and $\mathcal{M}$ be a marking of $\mathcal{NL}$. A finite sequence of steps $\rho = U_1 \dots U_n$, $n \in \mathbb{N}^+$ is called a step sequence enabled in $\mathcal{M}$ and leading to $\mathcal{M}_n$ if there exists a sequence of markings $\mathcal{M}_1, \dots, \mathcal{M}_{n-1}$ such that $\mathcal{M}[U_1\rangle\mathcal{M}_1[U_2\rangle \dots [U_n\rangle\mathcal{M}_n$.*

The marking $\mathcal{M}_n$ is reachable from the marking $\mathcal{M}$ if and only if there exists a step sequence enabled in $\mathcal{M}$ and leading to $\mathcal{M}_n$.

We can extended this definition to maximal step sequences. A finite sequence of maximal steps $\rho = U_1 \dots U_n$, $n \in \mathbb{N}^+$ is called a maximal step sequence enabled in $\mathcal{M}$ and leading to $\mathcal{M}_n$ if $\mathcal{M}[U_1\rangle_m\mathcal{M}_1[U_2\rangle_m \dots [U_n\rangle_m\mathcal{M}_n$.

A (maximal) step sequence $\rho = U_1 \dots U_n$ is *halting* if no non-empty (maximal) step is fireable at $\mathcal{M}_n$ and the marking $\mathcal{M}_n$ is called a terminal marking, A computation of a Petri net $\mathcal{NL}$ is an halting (maximal) step sequence starting from the initial marking and every marking appearing in such a sequence is called reachable. We shall write respectively $S(\mathcal{NL})$ and $S_m(\mathcal{NL})$ for the sets of halting step sequences and maximal step sequences of $\mathcal{NL}$ fireable at the initial marking $\mathcal{M}_0$. The step sequences whose steps consist of at most one transition are called firing sequences. We denote the set of halting firing sequences by $T(\mathcal{NL})$.

Step Languages of the Petri nets

Let $\mathcal{K} = (V, \mathcal{NL}, \zeta)$, $\mathcal{NL} = (P, T, F, W, G, \mathcal{M}_0)$, be a labelled Petri net, where

- V is an alphabet.
- $\mathcal{NL} = (P, T, F, W, G, \mathcal{M}_0)$ is a Petri net.
- $\zeta : 2^T/\{\emptyset\} \to V \cup \{\lambda\}$ defines the symbol-wise labelling for every step. A step $U \in 2^T/\{\emptyset\}$ is called λ-step, if $\zeta(U) = \lambda$.

To associate an SN P system language with the corresponding Petri net language we require that the sequence of applied rules corresponds to a step occurrence sequence of the Petri net.

Therefore as a correspondence we choose a weak coding (any transition is mapped to a symbol or the empty word) which agree with the classical variants of Petri net languages. We consider only one type of acceptance from the theory of Petri net languages: only those step occurrence sequences belonging to the languages which transform the initial marking into one of the terminal markings of $\mathcal{NL}$. As we are considering only the step occurrence sequence leading to a terminal marking, we say that the Petri net step language is of T-type.

Unlike associating a symbol to each transition of the Petri net as in [53], we associate a symbol to each step U of $\mathcal{NL}$. We define a labelling function ζ that maps each step with a symbol of a finite alphabet V. We are assuming that ζ is a free labelling function; i.e., step can be labelled with the empty symbol λ and several transitions may have the same label.

The mapping ζ will be extended to step sequences and maximal step sequences. Let $\zeta : S(\mathcal{NL}) \rightarrow V^*$. For a halting step sequence $\rho = U_1 \dots U_n \in S(\mathcal{NL})$, $\zeta(\rho)$ is the word $\zeta(U_1) \dots \zeta(U_n)$. The step language $L^s(\mathcal{NL})$ of $\mathcal{NL}$ is a set $L^s(\mathcal{NL}) = \{\zeta(\rho) \mid \rho \in S(\mathcal{NL})\}$. Similarly the maximal step languages are defined as $L^m(\mathcal{NL}) = \{\zeta(\rho) \mid \rho \in S_m(\mathcal{NL})\}$.
If firing sequences are considered then $L(\mathcal{NL}) = \zeta(T(\mathcal{NL}))$ is the standard T-type Petri net language of $\mathcal{NL}$.

4.4 Translating SN P Systems into Petri Nets

In this section, we propose a formal method to translate standard spiking neural P systems into Petri net models with step semantics.

To model a standard SN P system $\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, syn, i_0)$ as a Petri net, we introduce a different place p_i with n_i tokens for each neuron σ_i. The environment of the SN P system is represented using place p_{m+1}.

We add a synopsis $(i_0, m+1)$ to syn to allow the transitions corresponding to rules in output neuron to send tokens to the place corresponding to the environment. For each rule $\boldsymbol{ij} : E/a^r \rightarrow a; 0 \in R_i$, we introduce a transition t_{ij} together with an input place p_i and an arc of $W(p_i, t_{ij}) = r$. The output places of t_{ij} are all p_k such that $(i,k) \in syn$ with $W(t_{ij}, p_k) = 1$. The transitions corresponding to the forgetting rules are sink transitions with no output places. The regular expression E associated with the rule $\boldsymbol{ij} : E/a^r \rightarrow a; 0$ is translated into a guard function for t_{ij} that holds true if $\mathcal{M}(p_i) \in \Psi(L(E))$. For simulating the rules are without delay, we only need constant weighted arcs.

We add a synchronizing place p_{is} to each p_i that corresponds to a neuron having more than one rule so that only one transition to fire from input place p_i in each step. Input and output arcs of weight one are connected between each t_{ij} and p_{is}.

An augmented place p_{ia} is introduced for each neuron having a delayed rule which is initially kept empty. To mimic the working of the spiking rule with delay which is of the form $E/a^r \rightarrow a; d$ with $d \geq 1$, we introduce two transitions t_{ij}, $t_{ij'}$ and a place for delay p_{ijd} which is initially empty.

We place an incoming arc from place p_i to t_{ij} and an outgoing arc from t_{ij} to p_{ia} with the incoming arc expression defined as if $\mathcal{M}(p_i) - r > 0$ then $W(p_i, t_{ij}) = \mathcal{M}(p_i) - r$ else $W(p_i, t_{ij}) = 1$. The weight of the outgoing arc from t_{ij} to p_{ia} is $\mathcal{M}(p_i) - r$. The transition t_{ij} also has an incoming arc from place p_{is} and a guard that returns true only if $\mathcal{M}(p_i) \in \Psi(L(E))$. We also add an arc from t_{ij} to p_{ijd} of weight equal to the delay t. The presence of token in the delay place p_{ijd} indicates that the neuron σ_i is closed. So the markings of the places p_i and p_{ia} represents the number of spikes in the neuron σ_i when it is open and closed respectively. The number of tokens in place p_{ijd} is decreased by one in each step by a sink transition t_{ijd}. The transition t_{ijd}

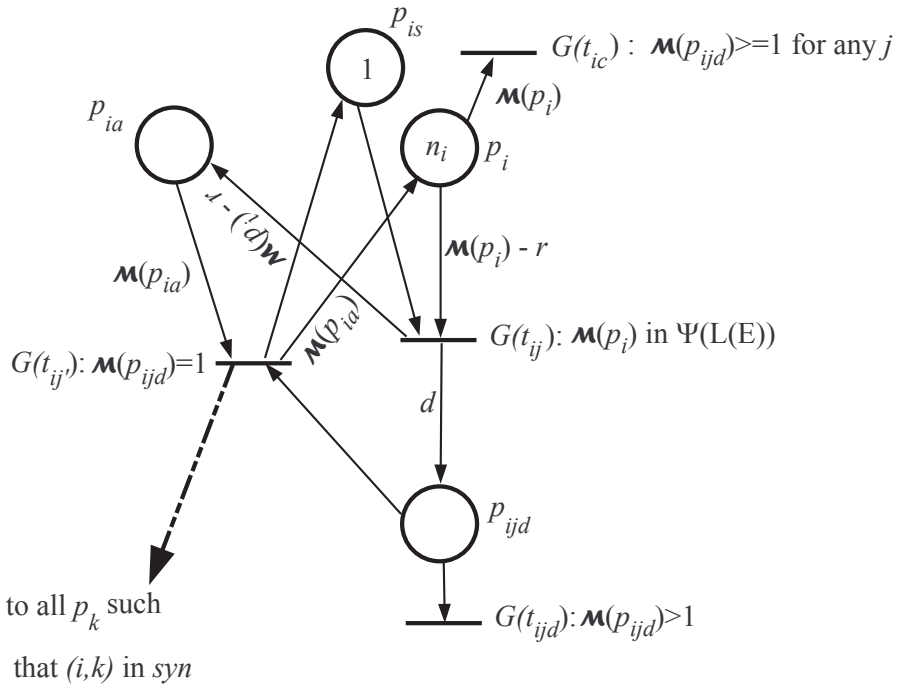


Figure 4.1: Sub net for the rule ***ij*** : $E/a^r \rightarrow a; d$

is not required when delay $d = 1$. When the contents of the delay place becomes one, the transition $t_{ij'}$ transfers back the contents of place p_{ia} to p_i and a token to all places p_k such that $(i,k) \in syn$ and to the place p_{is}. At the same time the contents of place p_i are cleared by the transition t_{ic}. So even though the tokens are collected in place p_i, its contents will be cleared only after the delay. Thus the translation mimics the working of the spiking rules with delay correctly.

The translation of the rule ***ij*** : $E/a^r \rightarrow a; d$ into a Petri net is shown in Figure 4.1. The label inside the place gives the initial marking of that place. Each transition is labelled with the guard function associated with it. In the construction described below, the SN P system is considered for the translation after adding a synopsis $(i_0, m+1)$ to syn. This adds an arc from output neuron to the environment.

Definition 4.7 (**SN P system to labelled Petri net**). *Let* $\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$,

syn, i_0) *be an SN P system, then the corresponding labelled Petri net is a construct* $\mathcal{K} \stackrel{df}{=} (V, \mathcal{NL}_\Pi, \zeta), \mathcal{NL}_\Pi = (P, T, F, W, G, \mathcal{M}_0)$, *where*

1. $V = \{0, 1\}$ *is an alphabet.*

2. *The components of* $\mathcal{NL}_\Pi$ *are defined as*

   (a) *The set of places is* $P \stackrel{df}{=} \{p_1, p_2, \ldots, p_m, p_{m+1}\} \cup \{p_{is} \mid R_i$ *has more than one rule* $\} \cup \{p_{ia} \mid \sigma_i$ *has at least one rule with delay* $d \geq 1\} \cup \{p_{ijd} /$ ***ij*** *is delayed rule in* $\sigma_i\}$.
   The initial marking of each place $\mathcal{M}_0(p_i) \stackrel{df}{=} n_i$ *and* $\mathcal{M}_0(p_{is}) = 1$ *for* $1 \leq i \leq m$, *while all other places are initially kept empty.*

   (b) *For each neuron* σ_i *and for each rule* ***ij*** $\in R_i$, *the set of transitions* T *contains a unique transition* $t' = t_{ij}$ *with the following connectivity.*

      i. *if* ***ij*** *is a forgetting rule of the form* $E/a^r \rightarrow \lambda$ *then* t' *will be a sink transition with* $W(p_i, t') = r, W(t', p_{is}) = 1$.

      ii. *if* ***ij*** *is a spiking rule of the form* $E/a^r \rightarrow a$ *then we have*
      $W(p_i, t') = r, W(t', p_{is}) = 1$ *and*
      $W(t', p_k) = 1$ *for every* $(i, k) \in syn$.

      iii. *if* ***ij*** *is a spiking rule of the form* $E/a^r \rightarrow a; d$ *with* $d \geq 1$ *then add a transition* $t_{ij'}$ *to* T *such that*
      $W(p_i, t') \stackrel{df}{=}$ *if* $\mathcal{M}(p_i) - r > 0$ *then* $\mathcal{M}(p_i) - r$ *else* 1,
      $W(t', p_{ia}) = \mathcal{M}(p_i) - r$,
      $W(p_{ia}, t_{ij'}) = \mathcal{M}(p_{ia}), W(t_{ij'}, p_i) = \mathcal{M}(p_{ia})$,
      $W(t', p_{ijd}) = d, W(p_{ijd}, t_{ij'}) = 1$ *and* $W(t_{ij'}, p_{is}) = 1$
      $G(t_{ij'}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{ijd}) = 1)$ *then return* $true$ *else return* $false$
      if $d > 1$ *then introduce a sink transition* t_{ijd} *to* T *with*
      $W(p_{ijd}, t_{ijd}) = 1$ *and*
      $G(t_{ijd}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{ijd}) > 1)$ *then return* $true$ *else return* $false$

iv. Set $W(p_{is}, t') = 1$

$G(t') \stackrel{df}{=}$ ***if*** $(\mathcal{M}(p_i) \in \Psi(L(E)))$ ***then return*** $true$ ***else return*** $false$

(c) The transition set T *also contains a transition* t_{ic} *for each neuron* σ_i *with a delayed rule having the following connectivity.*

$W(p_i, t_{ic}) = 1$

$G(t_{ic}) \stackrel{df}{=}$ ***if*** $(\mathcal{M}(p_{ijd}) = 1)$ ***then return*** $true$ ***else return*** $false$

3. $\zeta : 2^T/\{\emptyset\} \rightarrow V$ ***where*** $\zeta(U) = 1$ ***if*** $\exists t \in U$ ***such that*** $W(t, p_{m+1}) = 1$***, otherwise*** $\zeta(U) = 0$

To prove the equivalence of SN P system and the corresponding Petri net, we show that the languages generated by both the systems is same. To capture a very tight correspondence between the SN P system Π and the corresponding Petri net $\mathcal{NL}_\Pi$, we introduce a straightforward bijection between configurations of Π and the sub markings of $\mathcal{NL}_\Pi$, based on the correspondence between places and neurons.

Let $\mathcal{C} = \langle \alpha_1/d_1, \alpha_2/d_2, \ldots, \alpha_m/d_m \rangle$ be a configuration of the SN P system Π. The corresponding configuration mapped sub marking $\phi(\mathcal{C})$ of $\mathcal{NL}_\Pi$ is defined as $\phi(\mathcal{C}) \stackrel{df}{=} \langle \beta_1/f_1, \beta_2/f_2, \ldots, \beta_m/f_m \rangle$ where for $1 \leq i \leq m$,

$$\phi(\mathcal{C})(\beta_i) \stackrel{df}{=} \begin{cases} \mathcal{M}(p_i) & \textbf{\textit{if}}\ d_i = 0 \\ \mathcal{M}(p_{ia}) & \textbf{\textit{if}}\ d_i \geq 1 \end{cases}$$

$$\phi(\mathcal{C})(f_i) \stackrel{df}{=} \begin{cases} \mathcal{M}(p_{ijd}) & \textbf{\textit{if}}\ d_i \geq 1 \text{ and } \mathcal{M}(p_{ijd}) \geq 1 \text{ for any } j \\ 0 & \textit{otherwise} \end{cases}$$

Similarly, for any vector rule $\boldsymbol{v} = \langle \boldsymbol{1j}_1, \boldsymbol{2j}_2, \ldots, \boldsymbol{mj}_m \rangle$ of Π enabled at configuration $\mathcal{C}$, we define an enabled maximal step $\xi(\boldsymbol{v})$ of transitions of $\mathcal{NL}_\Pi$ such that $\xi(\boldsymbol{v}) \stackrel{df}{=} \{t_{ij_i} \mid \boldsymbol{v}(i) = \boldsymbol{ij}_i \textbf{\textit{ with}}\ j_i \geq 1, 1 \leq i \leq m\} \cup \{t_{ij_id} \mid \boldsymbol{v}(i) = \boldsymbol{ij}_i \textbf{\textit{ with}}\ j_i = 0 \textbf{\textit{ and}}\ d_i \geq 2, 1 \leq i \leq m\} \cup \{t'_{ij_i}, t_{ic} \mid \boldsymbol{v}(i) = \boldsymbol{is}, 1 \leq i \leq m\}$. It is clear that ϕ is a bijection from the

configurations of Π to the configuration mapped sub markings of $\mathcal{NL}_\Pi$, and that ξ is a bijection from vector rules of Π to enabled maximal steps of $\mathcal{NL}_\Pi$.

We now can formulate a fundamental property concerning the relationship between the dynamics of the SN P system Π and that of the corresponding Petri net $\mathcal{NL}_\Pi$:

$\mathcal{C} \stackrel{\boldsymbol{v}}{\Longrightarrow} \mathcal{C}'$ *if and only if* $\phi(\mathcal{C})[\xi(\boldsymbol{v})\rangle_m \phi(\mathcal{C}')$.

By the construction of Petri net, the initial configuration of Π corresponds through ϕ to the initial configuration mapped sub marking of $\mathcal{NL}_\Pi$. It can be observed that the structure of neurons in Π is used in the definitions of the structure of the net $\mathcal{NL}_\Pi$ (i.e., in the definitions of places, transitions, the weight function, and the guard function). Let $\mathcal{C}$ be a configuration of Π and there is a vector rule $\boldsymbol{v}$ enabled at $\mathcal{C}$ reaching a configuration $\mathcal{C}'$. As there is a mapping between configurations and sub markings, $\phi(\mathcal{C})$ is the marking of net $\mathcal{NL}_\Pi$ corresponding to the configuration $\mathcal{C}$ of Π. For locally sequential and globally maximal firing semantics of SN P system, there is a synchronizing place p_{is} for each place p_i, $1 \leq i \leq m$ to allow at most one transition (that too only one occurrence) to fire from p_i. There is a one-to-one mapping between the vector rule in the SN P system and maximal step in the corresponding Petri net. So there exists a maximal step $\xi(\boldsymbol{v})$ enabled at the sub marking $\phi(\mathcal{C})$. After the execution of $[\xi(\boldsymbol{v})\rangle_m$, the system reaches the configuration $\phi(\mathcal{C}')$. We can prove only if part in the similar way.

We now extend the statement for sequences of transitions and sequences of steps.
$\gamma = \mathcal{C}_0 \stackrel{\boldsymbol{v}_1}{\Rightarrow} \mathcal{C}_1 \stackrel{\boldsymbol{v}_2}{\Rightarrow} \ldots \stackrel{\boldsymbol{v}_k}{\Rightarrow} \mathcal{C}_k$ is an halting computation of Π *if and only if* $\Im(\gamma) = \phi(\mathcal{C}_0)[\xi(\boldsymbol{v}_1)\rangle_m \phi(\mathcal{C}_1)[\xi(\boldsymbol{v}_2)\rangle_m \ldots [\xi(\boldsymbol{v}_k)\rangle_m \phi(\mathcal{C}_k)$ is the halting maximal step sequence of $\mathcal{NL}_\Pi$.

So the evolution of the Petri net $\mathcal{NL}_\Pi$ is same as the evolution of the SN P system Π. That means $\gamma \in COM(\Pi)$ iff $\Im(\gamma) \in S_m(\mathcal{NL}_\Pi)$.

Let $\mathcal{C}_{i-1} \stackrel{\boldsymbol{v}_i}{\Rightarrow} \mathcal{C}_i$ is the ith step of γ and if $bin(v_i) = 1$. By the definition of bin, $bin(v_i) = 1$ iff $\boldsymbol{v}_i(i_0)$ is a non delayed spiking rule with $rhs(\boldsymbol{v}_i(i_0)) = 1$ or $\boldsymbol{v}_i(i_0) = \boldsymbol{is}$. From the construction of Petri net and the definition of $\xi(\boldsymbol{v}_i)$, we observe that the step $\xi(\boldsymbol{v}_i)$ contains a transitions t with $W(t, p_{m+1}) = 1$, which implies that $\zeta(\xi(\boldsymbol{v}_i)) = 1$. Similarly we can prove that $bin(\boldsymbol{v}_i) = 0$ iff $\zeta(\xi(\boldsymbol{v}_i)) = 0$. We extend this to the words generated by both systems. If $w = bin(\gamma) \in \{0,1\}^*$ iff $w = \zeta(\Im(\gamma))$.

From the above statement, we prove that $L(\Pi) = L^m(\mathcal{NL}_\Pi)$.

4.4.1 The Properties of SN P Systems Derived from Petri Nets

Many useful behavioural properties such as reachability, boundedness, liveness of Petri nets have been investigated. We also introduce these properties for SN P systems.
For a SN P system, we define structural analysis which can identify properties that are conserved during execution of the modelled system. It may provide insights to the system. Such properties include the following:

1. **Boundedness:** An SN P system is said to be k-bounded or simply bounded if the number of spikes in each neuron for every reachable configuration will not exceed a finite number k. It is checking that there is no infinite accumulation of spikes in a particular neuron.

2. **T-Invariants:** Identifying the sequence of vector rules that have to fire from the some initial configuration to return the SN P system to that configuration. T-invariants indicate the presence of cycles that are in a state of continuous operation.

3. **Reachability:** Deciding whether a certain configuration (state) is reachable from another configuration. This type of analysis can be used to determine whether certain outcomes are possible, given a modelled SN P system and an initial configuration (initial state), or to determine whether certain configurations are reachable when specific rules are inhibited.

4. **Terminating**: The sequences of transitions between configurations of a given SN P system is finite, i.e., the computation of the SN P system always halts.

5. **Deadlock-free**: Each reachable configuration enables a next transition.

6. **Liveness**: It is deadlock-free and there is a sequence of enabled vector rules.

Theorem 4.1. *If the Petri net for a given SN P system* Π *is terminating, then the SN P system* Π *is terminating.*

Proof. If the SN P system is not terminating, according to the definition of termination for SN P systems, there exists an infinite sequence of transitions. When the SN P system is encoded by the Petri net, there also exists an infinite step sequence. Every transition is one-to-one mapped to a step in the Petri net, so the sequence of steps in the Petri net is not finite. Thus, this Petri net is not terminating. □

Theorem 4.2. *If the Petri net for a given SN P system* Π *is deadlock-free, then the SN P system* Π *is deadlock-free.*

Theorem 4.3. *If the Petri net for a given SN P system* Π *has liveness, then the SN P system* Π *has liveness.*

Theorem 4.4. *If the Petri net for a given SN P system* Π *is bounded, then the SN P system* Π *is bounded.*

Proof. The proofs of Theorem 4.2, Theorem 4.3, and Theorem 4.4 are the same as for Theorem 4.1. □

4.5 Some Examples

Example 4.1.

First we consider an example of SN P system without delay. As all rules are having no delay, we represent the configuration of an SN P system without delay as $\langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ where neuron σ_i contains $\alpha_i \geq 0$ spikes, for $i = 1, 2, 3, \ldots, m$. Figure 4.2(a) represents the initial configuration of the SN P system Π_2. We have three neurons, labelled with $1, 2, 3$, with neuron σ_3 being the output one. The neurons are represented by nodes of a directed graph whose arrows represent the synapses; an arrow also exits from the output neuron, pointing to the environment; in each neuron we specify the rules which are labelled and the spikes present in the initial configuration. The rule ***11*** : $a^2/a \rightarrow a$ in σ_1 fires only if σ_1 contains two spikes; one spike is consumed, the other remains available for the next step. The rule ***12*** : $a^2 \rightarrow a$ also fires only if it contains two spikes; both are consumed. So in σ_1, there is a non-determinism between its two rules. Neuron σ_2 is having only one rule, and neuron σ_3 is having one firing and one forgetting rule. The SN P system Π_2 is formally represented as:

Π_2 = ($\{a\}$, σ_1, σ_2, σ_3, syn, 3), with

σ_1 = (2, $\{a^2/a \rightarrow a, a^2 \rightarrow a\}$),

σ_2 = (1, $\{a \rightarrow a\}$),

σ_3 = (1, $\{a \rightarrow a, a^2 \rightarrow \lambda\}$),

syn = {(1,2),(2,1),(1,3),(2,3)}.

The initial configuration of the system is $< 2, 1, 1 >$. It works as follows. All neurons can fire in the first step, with neuron σ_1 choosing non-deterministically between its two rules.

Output neuron σ_3 sends its spike to the environment. If the neuron σ_1 uses its

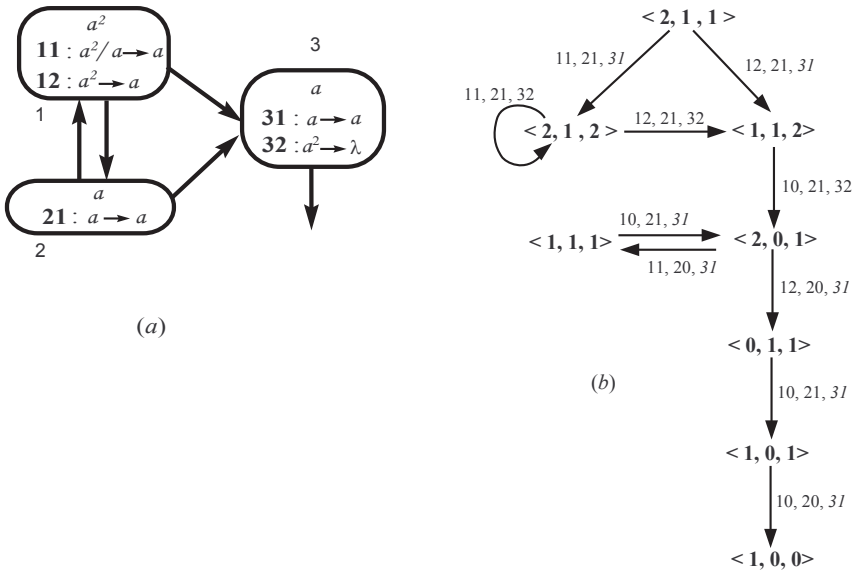


Figure 4.2: A Spiking neural P system Π_2 and its evolution

first rule, then both the neurons σ_1 and σ_2 exchange their spikes and send a spike to the output neuron σ_3 and reach the configuration $< 2, 1, 2 >$. As long as neuron σ_1 uses the rules $a^2/a \to a$, the computation cycles in the same configuration: neurons σ_1 and σ_2 exchange their spikes, while neuron σ_3 forgets its two spikes.

However, at any moment, starting with the first step of the computation, σ_1 can choose to use the rule $a^2 \to a$. This means that the two spikes of σ_1 are consumed and a spike sent to σ_2 and σ_3; in this way, σ_1 will have only one spike in the next step and the system reaches the configuration $< 1, 1, 2 >$. Here the neuron σ_1 cannot use any of its rule, while neuron σ_3 forgets its two spikes reaching the configuration $< 2, 0, 1 >$. Using the rules in this way the system reaches the halting configuration $< 1, 0, 0 >$.

The evolution of the system Π_2 can be analyzed on a transition diagram as that from Figure 4.2(b): because the system Π_2 is finite, the number of configurations reachable from the initial configuration is finite, too, hence, we can place them in the nodes of a graph, and between two nodes/configurations we draw an arrow if and only if a direct transition is possible between them. In Figure 4.2(b) we have also

indicated the rules used in each neuron, with the following conventions: ij denotes the jth rule in neuron σ_i, with *31* being written in italics, in order to indicate that a spike is sent out of the system at that step; when a neuron σ_i, $i = 1, 2, 3$ uses no rule, we have written $i0$.

The transition diagram of a finite SN P system can be interpreted as the representation of a non-deterministic finite automaton, with $\mathcal{C}_0$ being the initial state, the halting configurations being final states, and each arrow being marked with 0 if in that transition the output neuron does not send a spike out, and with 1 if in the respective transition the output neuron spikes; in this way, we can identify the language generated by the system. In case of SN P system Π_1, language generated is $L(\Pi_2) = L(\mathbf{10}^+(\mathbf{11})^*\mathbf{111})$.

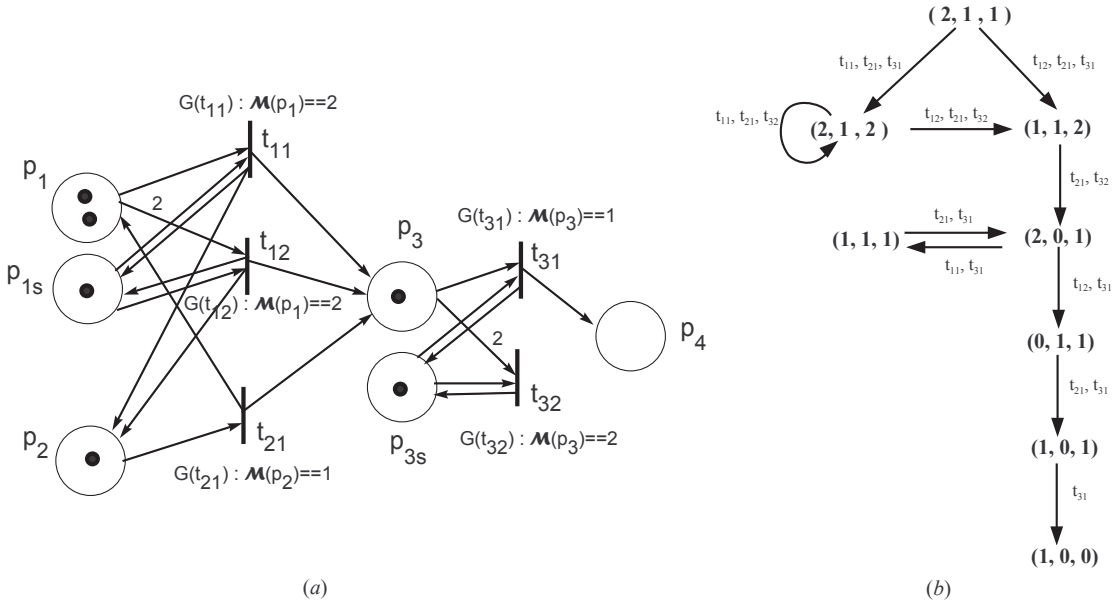


Figure 4.3: Petri net $\mathcal{NL}_{\Pi_2}$ equivalent to the SN P system Π_2

Figure 4.3(a) gives the Petri net representation of the SN P system Π_2. p_1, p_2, p_3, and p_4 are the places corresponding to neurons σ_1, σ_2, σ_3, and environment of Π_2 respectively. We do not need any delay places here as all rules are without delay. The place p_2 do not require any synchronizing place as the corresponding neuron σ_2

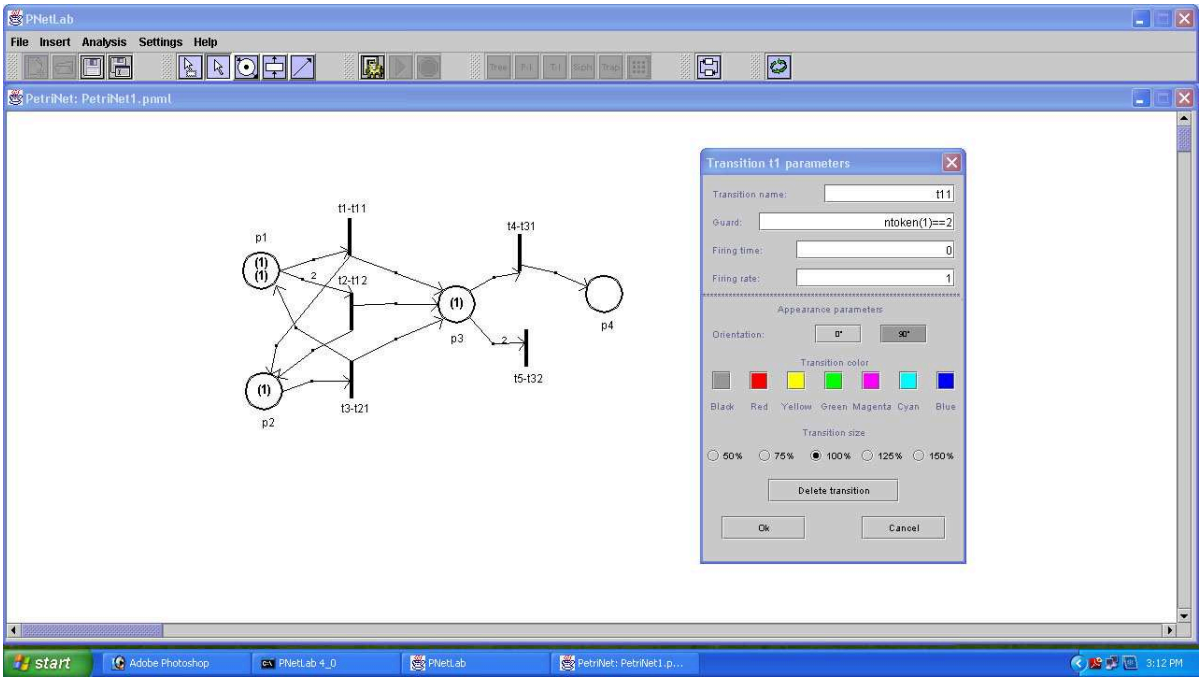
has only one rule and synchronizing places are introduced for the other two places p_1 and p_3 with a token in each. The places p_1, p_2, and p_3 are initially marked with 2, 1, and 1 token respectively which is same as the initial numbers of spikes in their corresponding neurons in Π_2. For each rule ***ij*** in Π_2, a transition t_{ij} is introduced with an incoming arc from place p_i and outgoing arcs to all places p_k such that $(i,k) \in syn$. The weights of the arcs are constants as in case of P/T nets and are not marking-dependent. A guard function is associated with each transition t_{ij} corresponding to the regular expression associated with ***ij***. Submarking reachability tree of the places p_1, p_2 and p_3 is given in Figure 4.3(b), where arcs are labelled with the maximal steps. We can observe from Figure 4.3(a) that t_{31} is the only transition having an arc to the place p_4, which corresponds to the environment of SN P system Π_2. By using labelling function ζ defined in the previous section, if we label the steps having t_{31} as 1 and other steps as 0, we get the step languages generated by the Petri net as $L^m(\mathcal{NL}_{\Pi_2}) = L(\mathbf{10}^+(\mathbf{11})^*\mathbf{111})$= $L(\Pi_2)$.

Simulation with PNetLab

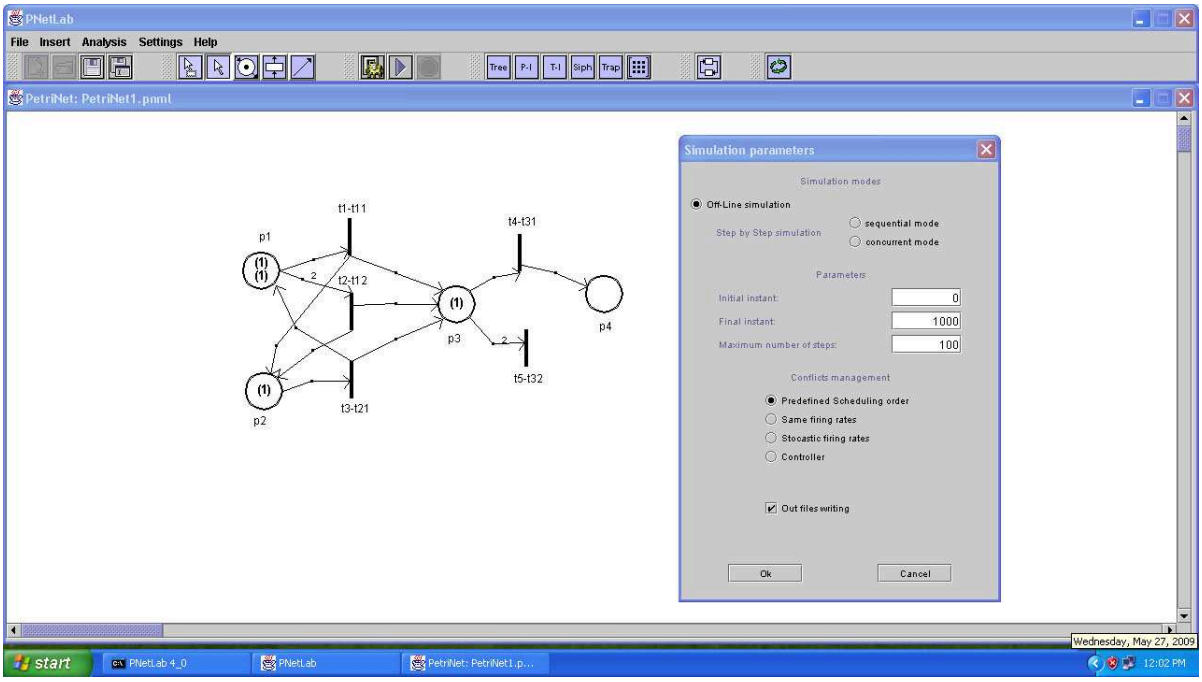
The previous section provides the translation of SN P systems into Petri nets that can be simulated using any tool that allows parallel execution of transitions. Here we consider PNetLab to model $\mathcal{NL}_{\Pi_2}$ obtained in the previous section after translation.

In each step, the tool allows only one transition to fire from each input place so synchronizing place is not required when the Petri net model is simulated through PNetLab.

Here we make use of the built-in function $ntoken(i)$ that returns the marking of the place p_i i.e. $\mathcal{M}(p_i)$. To implement the regular expression E associated with each rule, a deterministic finite automata for E is constructed and is translated into a user defined guard function that enables the transition corresponding to the rule when the number of tokens present in the input place p_i is in $\Psi(L(E))$.



(a)



(b)

Figure 4.4: PNetLab model for the Petri net $\mathcal{NL}_{\Pi_2}$

A central reason for defining Petri net models for SN P systems is that there are variety of analytical and verification techniques developed for Petri nets which could be applicable to SN P systems. For the purpose of analysis of an SN P systems we can also investigate the reachability graph of its Petri net as this is isomorphic to the reachability graph of the SN P systems. Since reachability graph combine step sequences and reachable states (markings), they are useful for the analysis and verification of behavioural properties.

Figure 4.4(a) shows the PNetLab model for the Petri net in Figure 4.3(a). The number of (1)'s inside the place indicates the marking of that place. Each transition t_{ij} of $\mathcal{NL}_{\Pi_2}$ is named as $tl - tij$, where tl is the transition name given by the tool. Each place p_i is named as pi. Figure 4.4(a) also gives guard function for the transition $t1 - t11$. If we consider the sub marking for the first three places $p1$, $p2$ and $p3$ (the places corresponding to the neurons), the initial sub marking is $< 2, 1, 1 >$ which is similar as that of the SN P system in Figure 4.2. Figure 4.4(b) gives how to set the simulation parameter and conflict managements, if more than one transition is enabled from the same input place.

Figure A.1$(a)-(h)$ in Appendix A gives the output of the step-by-step simulation of the model. The small window displayed in the bottom right corner shows the step number and the transitions (transition names are as given by the tool) fired in the step. In the step 1, after the firing of the transitions $t1 - t11, t3 - t21, t4 - t31$ (corresponding to rules ***11***, ***21***, ***31*** of Π_2), the system reaches the same sub marking $< 2, 1, 2 >$ (shown in Figure A.1(a) of Appendix A). It stays in the same marking as long as the step $t1 - t11, t3 - t21, t5 - t32$ is fired. When the system chooses the transition $t2 - t12$ instead of $t1 - t11$ in the step 2, it reaches the configuration $< 1, 1, 2 >$ (see in Figure A.1(b)). Figure A.1$(c) - (h)$ gives remaining steps of the execution.

Figure 4.5 gives the report of markings during the simulation. The figure is the modified part (some Italian words are translated to English) of the *statomarche.xml* file which is generated by the tool during the simulation of the model and is found in

Figure 4.5: Report of markings of $\mathcal{NL}_{\Pi_2}$ in the steps of simulation in PNetLab

the directory *∼/PNetLab 4.0/Engine/SimEngine/OutXML*. Each tables gives the marking of the places during each pass (or step). The number of $\{1\}$'s in front of the place indicates the marking of the place.

We can observe from Figure 4.5 that the configurations reachable from the initial configuration of the SN P system Π_2 are same as the markings reachable from the initial marking in the corresponding Petri net model. So we conclude that the Petri net model in Figure 4.4(a) accurately simulates the working of the SN P system Π_2.

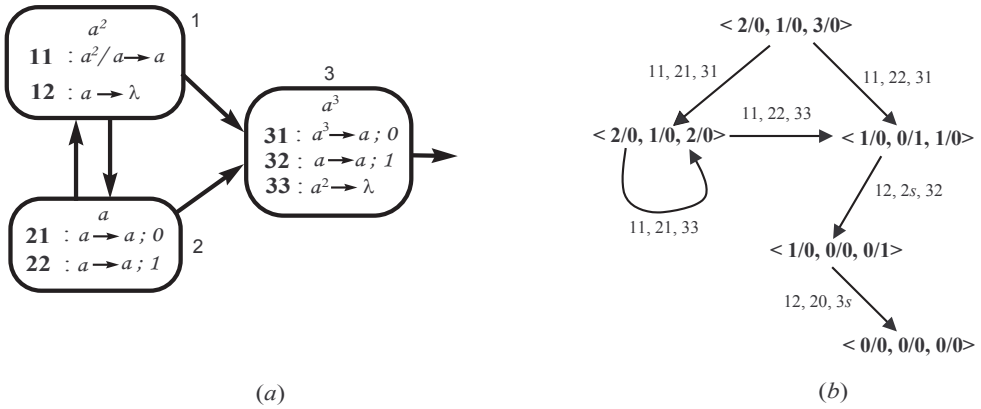
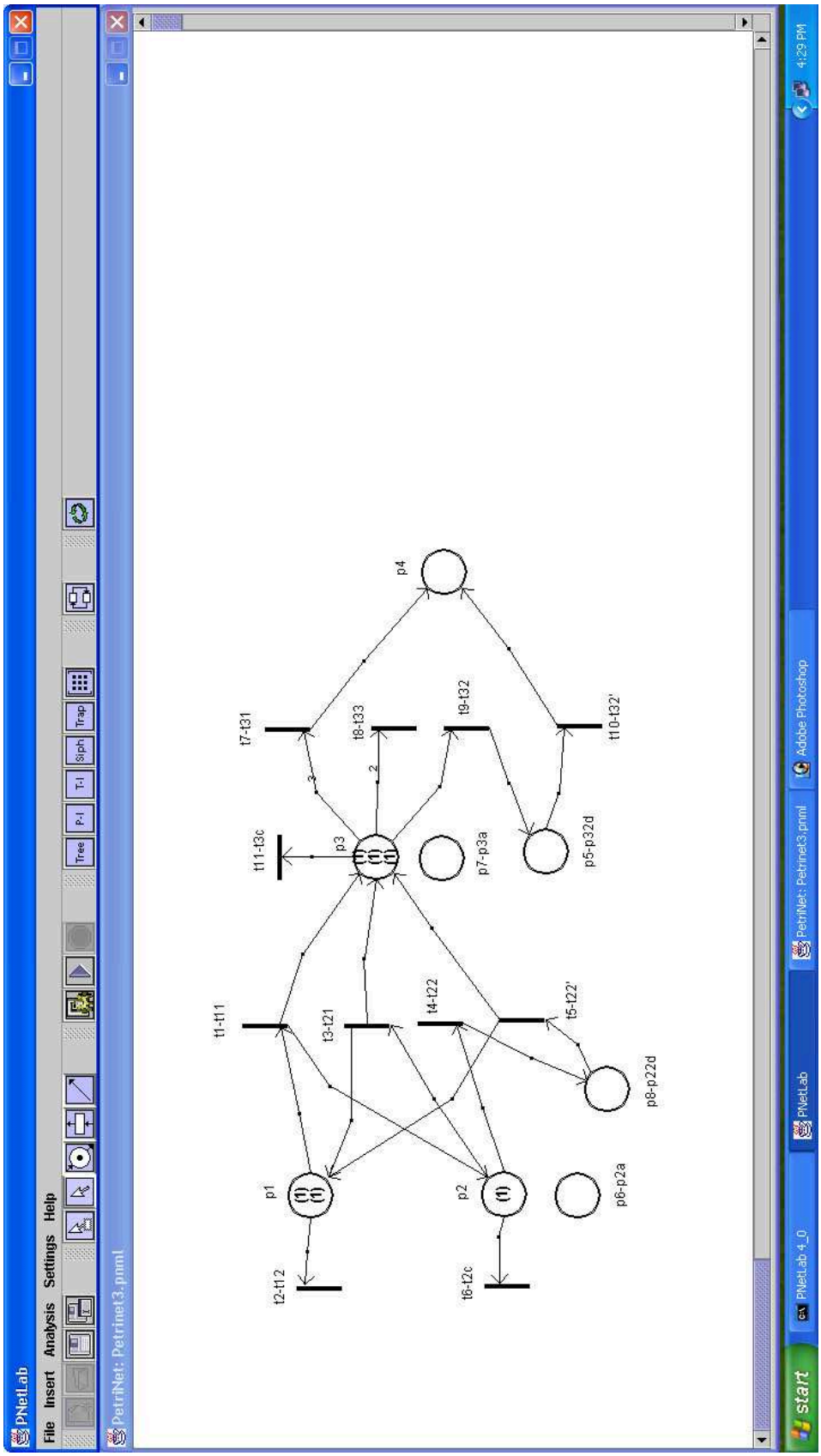


Figure 4.6: (a) An SN P system Π_1 (b) Evolution of Π_1

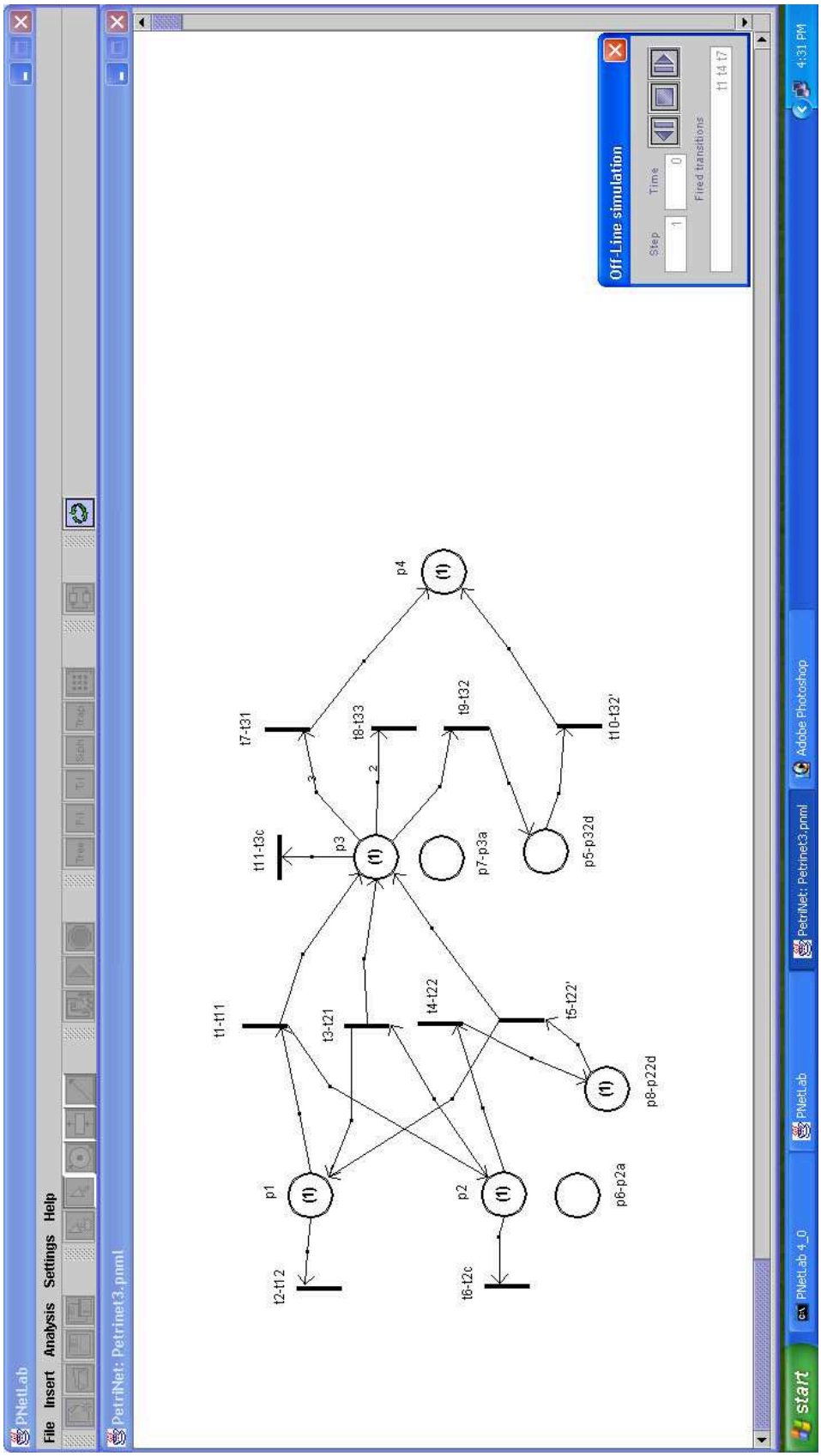
Example 4.2.

Here we simulate the SN P system with delay Π_1 discussed in Example 2.2 of Chapter 2. The graphical representation of SN P system is reproduced in Figure 4.6 for the readability.

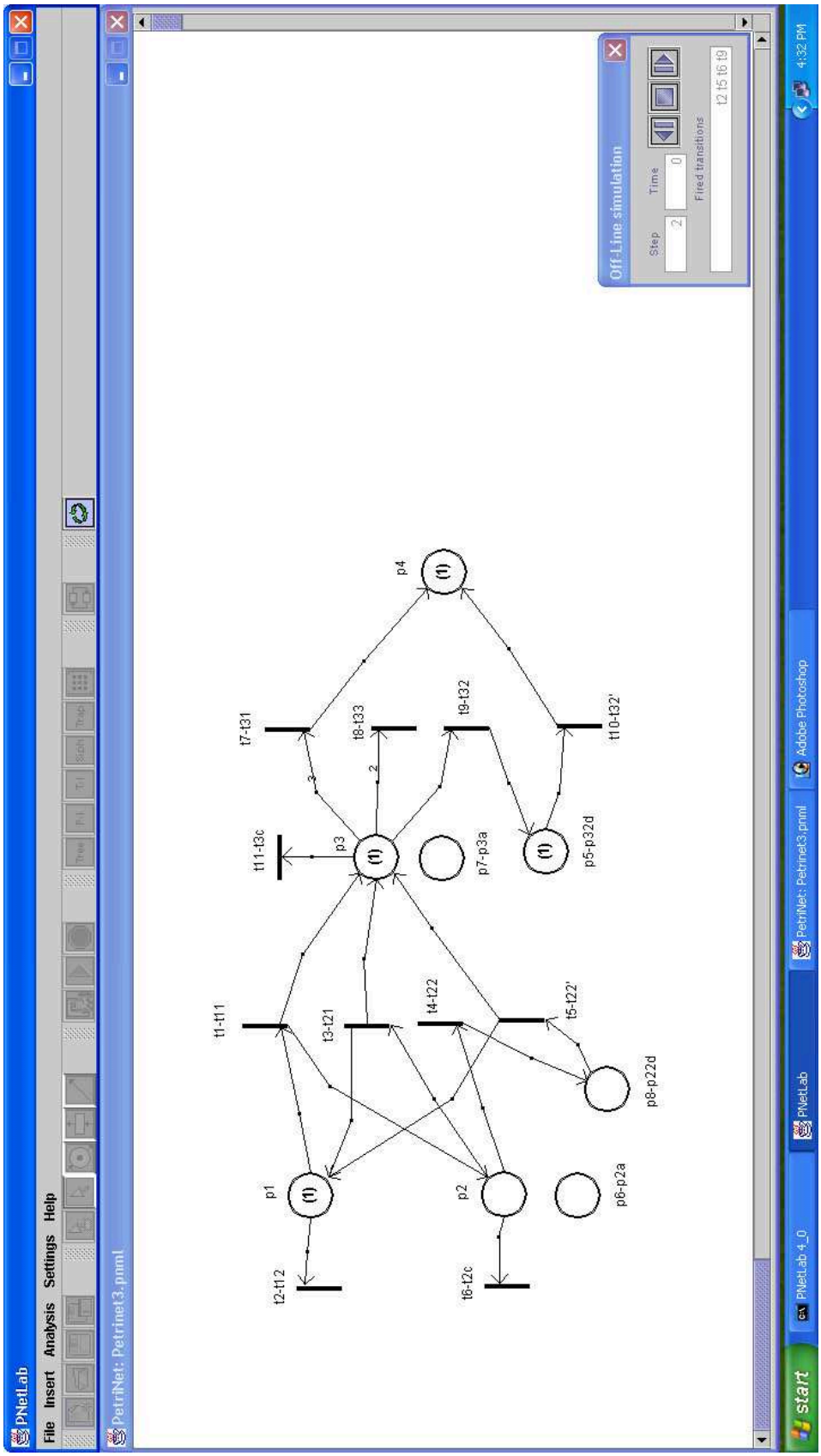
Here we directly model the SN P system in PNetLab. Figure 4.7$(a) - (d)$ shows the PNetLab model for the SN P system Π_1 and its execution in step-by-step mode. Each transition is named as $tl - tij$, where tl is the transition name given by the tool and tij corresponds to transition name t_{ij} given as per methodology discussed in the previous section. p_1, p_2, p_3 and p_4 are the places corresponding to the neurons σ_1, σ_2, σ_3 and the environment respectively. The other places are named in a similar way as the transitions. The symbol (1) in side the place indicates the presence of a token. The transition $t1 - t11$ corresponds to the spiking rule ***11*** : $a^2/a \rightarrow a; 0$ of the neuron σ_1. The regular expression associated with the rule ***11*** is implemented by associating guard function $ntoken(1) == 2$ with $t1 - t11$, which allows the transition to fire only if the place $p1$ has exactly two tokens. Since the neuron σ_1 has an out going synapses



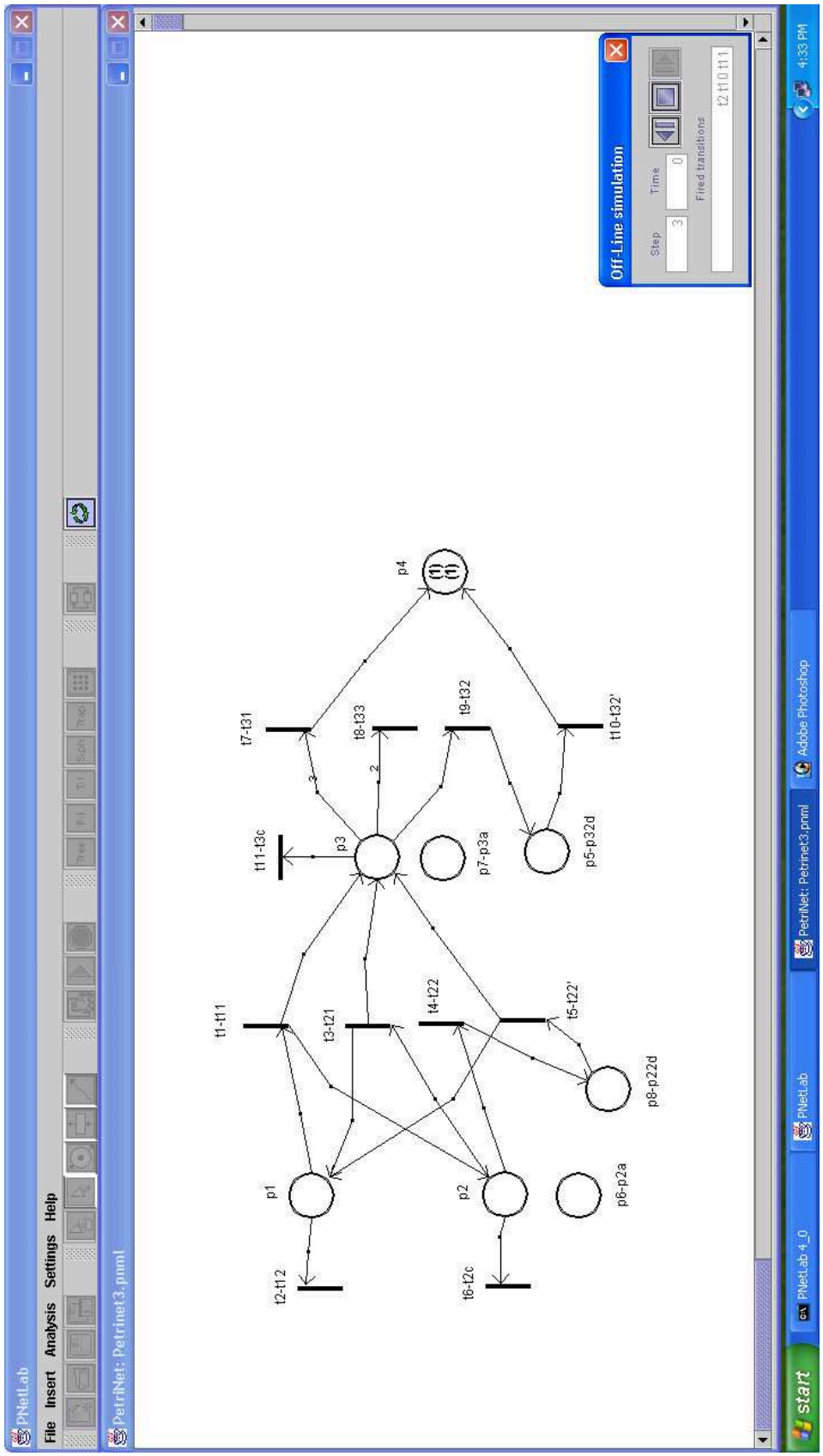
(a) Simulation of $\mathcal{NC}_{\Pi_1}$ in PNetLab (Step 1)



(b) Simulation of $\mathcal{NC}_{\Pi_1}$ in PNetLab (Step 2)



(c) Simulation of $\mathcal{NC}_{\Pi_1}$ in PNetLab (Step 3)



(d) Simulation of $\mathcal{NC}_{\Pi_1}$ in PNetLab (Step 4)

Figure 4.7: Step-by-step simulation of $\mathcal{NC}_{\Pi_1}$ in PNetLab

to other two neurons, the transition $t1 - t11$ has an input arc of weight one from place $p1$ and two outgoing arcs of unit weight to places $p2$ and $p3$. The forgetting rule of σ_1 is translated into a sink transition $t2 - t12$ with an incoming arc from $p1$. It is associated with the guard $ntoken(1) == 1$. The other spiking rules without delay and forgetting rules of σ_2 and σ_3 are translated in the similar way.

To implement the rule ***22*** $: a \rightarrow a; 1$ of σ_2 in PNetLab, two places $p6 - p2a$, $p8-p22d$ and two transitions $t4-t22$ and $t5-t22'$ are introduced with $W(p2, t4-t22) = 1$, representing the transfer of a spike from neuron σ_2 and $W(t4 - t22, p8 - 22d) = 1$, describing the delay $d = 1$. The place $p8 - p22d$ gives the time after which the neuron σ_2 becomes open. The transitions $t4-t22$ is associated with a guard function $ntoken(2) == 1$, which fires when the place $p2$ has exactly one token, which conflicts with the guard function of $t3 - t21$. So at any time only one of the transitions $t4 - t22$ or $t3 - t21$ will fire, which leads to a non-determinism. There is no arc between $t4 - t22$ and $p6 - p2a$ as no tokens will be left in place $p2$ after firing of the transition $t4 - t22$. The delay place $p8 - p22d$ has an outgoing arc to $t5 - t22'$, which when fires transfers a token to places $p1$ and $p3$ (Since σ_2 has an out going synapses to σ_1 and σ_3). The other spiking rule with delay ***32*** $: a \rightarrow a; 1$ of σ_3 is also implemented in a similar way.

Figure 4.8 gives the report of markings of the step-by-step simulation of the model in PNetLab. The figure is the part of the statomarche.xml file which is generated during the simulation of the model and is found in the directory ∼/PNetLab 4.0/Engine/ SimEngine/OutXML. The markings of the places p_i and p_{ia} corresponds to the number of spikes in the neuron σ_i when it is open and closed respectively. The configuration mapped sub marking of the Petri net $\mathcal{NL}_{\Pi_1}$ is $\langle \beta_1/d_1, \beta_2/d_2, \beta_3/d_3 \rangle$ where β_i and d_i are calculated as per the procedure discussed in the previous section. So the initial sub marking is $\langle \mathcal{M}(p_1)/0, \mathcal{M}(p_2)/0, \mathcal{M}(p_3)/0 \rangle$= $\langle 2/0, 1/0, 3/0 \rangle$, which is similar to the initial configuration of the SN P system Π_1 in Figure 4.6. In the first pass (step), $t4 - t22$ is chosen instead of $t3 - t21$, which corresponds to the rule ***22*** $: a \rightarrow a; 1$.

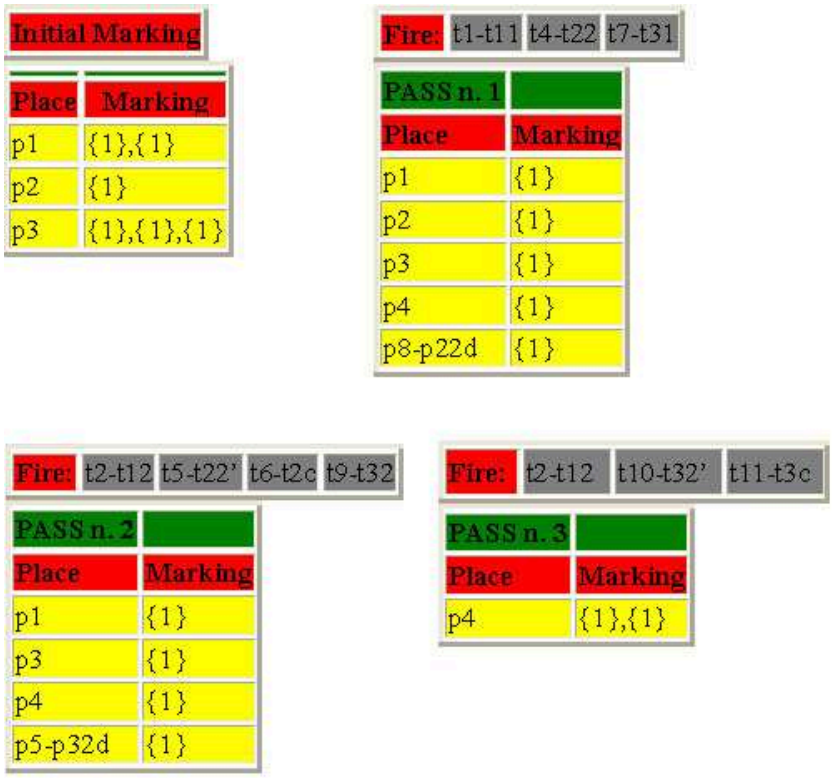


Figure 4.8: Report of markings of $\mathcal{NL}_{\Pi_1}$ in the steps of simulation in PNetLab

It transfers a token to the place $p8-p22d$ which represents that the second neuron is closed and will open after one time unit. The contents of the place $p6-p2a$ gives the number of spikes in the second neuron. So after firing of the step containing transitions $t1-t11, t4-t22, t7-t31$, the next configuration mapped sub marking of the Petri net $\mathcal{NL}_{\Pi_1}$ is $\langle \mathcal{M}(p_1)/0, \mathcal{M}(p6-p2a)/\mathcal{M}(p8-p22d), \mathcal{M}(p_3)/0\rangle$=$\langle 1/0, 0/1, 1/0\rangle$. In the next step transitions $t2-t12, t5-t22', t6-t2c, t9-t32$ (corresponding to rules ***12***, ***2s***, ***32***) are enabled. The transition $t9-t32$ corresponds to the delayed rule of σ_3. The transition $t6-t2c$ clears the contents of place $p2$. So upon firing this step, the system reaches the next sub marking $\langle 1/0, 0/0, 0/1\rangle$. The transitions enabled at this marking are $t2-t12, t10-t32', t11-t3c$ reaching the final marking $\langle 0/0, 0/0, 0/0\rangle$. We can observe from the Figure 4.8 that the configurations reachable from the initial configuration of the SN P system are same as the configuration mapped sub markings reachable in the corresponding Petri net model from the initial sub marking.

So we conclude that the Petri net model in Figure 4.7$(a) - (d)$ accurately simulates the working of the SN P system Π_1 in Figure 4.6.

4.6 Conclusion

In this chapter we demonstrated that Petri net is a formal model that gives a representation of an SN P system which is sufficiently rich to allow one to represent properties and aspects of the system which may be relevant for the design and verification activities. Besides representing the state and the architectural aspects of a system, Petri net typically comes equipped with an operational semantics which can formally explain how the system behaves.

The chapter provides a systematic procedure to translate standard SN P systems into Petri nets that can be simulated using any tool that supports parallel execution of transitions and guard functions. Here we consider a tool called PNetLab to simulate Petri net models.

In standard SN P systems the firing mode is the maximal strategy. In this mode, in each transition only one rule is applied from each neuron that also at most once. If in a configuration there are different sets of rules that can be applied in a neuron, then one of them is non-deterministically chosen.

The sequential SN P systems and asynchronous SN P systems differ from the standard SN P systems only in the mode of execution. So the procedure to model these SN P systems is same as the procedure discussed in this chapter. But in order to simulate the sequential SN P systems, we can consider the sequential mode of execution for Petri nets. To simulate the asynchronous SN P systems, we can consider the step execution semantics of the Petri nets.

We can also simulate extended SN P systems, which allows the rules are of the form $E/a^r \rightarrow a^q; t$, where $r \geq 1$ and $q \geq 0$. This means, if the neuron σ_i has number

of spikes in $\Psi(L(E))$, then r spikes are consumed and q are produced and sent to the neurons to which there exist synapses leaving the neuron where the rule is applied. The rule is implemented in a similar way as the spiking rule of SN P system shown in Figure 4.1 but unit arcs from transition $t_{ij'}$ to all p_k with $(i,k) \in syn$ (shown as dotted arc in the figure) are replaced with an arc weight of q, which represents the transfer q tokens.

Our intention is to go further on the line of this research to do thorough investigation of the Petri net model considered for translation, to relate this model to the known variants of Petri nets, and to study what behavioural problems are decidable in this framework.

Chapter 5

SN P Systems with Anti-Spikes and Petri Nets

Spiking neural P systems with anti-spikes works in the same way as standard SN P system but deals with two types of objects called spikes (a) and anti-spikes ($\overline{a}$). There is also an highest priority annihilation rule ($a\overline{a} \rightarrow \lambda$) that is implicitly present in each neuron of an SN PA system. So it is challenging to translate these systems into Petri net models that can be simulated using a Petri net tool. In this chapter we propose a methodology to model and simulate SN PA systems using Petri nets. This enables us to verify system properties, system soundness and to simulate the dynamic behaviour. We start with Section 5.2 by giving a brief introduction about SN PA systems. A procedure to translate SN PA systems into an equivalent Petri net models is provided in Section 5.3. In Section 5.4, the translation procedure is illustrated with an example and analysis results for the SN PA system is studied by simulating and analysing the obtained Petri net model in PNetLab.

5.1 Introduction

In a standard SN P system there are only one type of objects called spikes which are moved, created and destroyed but never modified into another form. SN P system with anti spikes (shortly called SN PA system) introduced in [79], is a variant of an SN P system consisting of two types of objects, spikes (denoted as a) and anti-spikes (denoted as $\overline{a}$). The inhibitory impulses/spikes are represented using anti-spikes. The anti-spikes behave in a similar way as spikes by participating in spiking and forgetting rules. They are produced from usual spikes by means of usual spiking rules; in turn, rules consuming anti-spikes can produce spikes or anti-spikes (here we avoid the rule anti-spike producing anti-spike). Each neuron in the system consists of an implicit annihilation rule of the form $a\overline{a} \rightarrow \lambda$; if an anti-spike and a spike meet in a given neuron, they annihilate each other. This rule has the highest priority and does not consume any time. So at any instant of time, a neuron in an SN P system with anti-spikes can have spikes or anti-spikes but not both.

The initial configuration of the system is described as the initial number of spikes or anti-spikes present in each neuron. The SN PA system evolves in a synchronous fashion, meaning that a global clock is assumed and in each time unit all neurons work in parallel with each neuron which can use a rule should do it, but using only one rule at a time (sequential locally). Using the rules, we can define transitions among configurations. The sequences of transitions among configurations, starting from initial configuration is called a computation. A computation halts if it reaches a configuration where no rule can be used. With any computation whether halting or not together with output produced in such case, yielding notions of functionality and computational power of SN PA systems including various aspects of computing.

It is extremely important to simulate these models to portray the system behaviour. Such models can shed insight into complex processes and suggest new

directions for research. Scientists can study and analyze such models to make predictions about the behaviour of the system under different conditions and to discuss novel relationships among the different components of a system. The ability to predict system behaviour with a model helps to evaluate model completeness as well as improve our understanding of the system.

A modelling methodology that is especially tailored for representing and simulating parallel dynamic systems is Petri nets. An advantage of Petri nets is that they have a visual representation and simulation that facilitates user comprehension. Petri net tools enable users to verify system properties, verify system soundness, and to simulate the dynamic behaviour.

In this chapter, we introduce the direct translation of SN PA systems into Petri nets models that can be simulated using existing Petri net tools. As the procedure is direct, it involves less complexity in translation and also using the notions and tools already developed for Petri nets, one can describe the internal process occurring during a computation in the SN PA system in a graphical way. Perhaps the greatest advantages of Petri nets are a solid mathematical foundation and the large number of techniques being developed for their analysis. These include: reachability analysis, invariants analysis (a technique using linear algebra), transformations (including reductions) preserving desired properties, structure theory and formal language theory.

5.2 SN P Systems with Anti-Spikes

An SN PA system works in a similar way as that of standard system without delay but deals with two types of objects. The mathematical definition of the SN PA systems is given in Section refsec:snpa. The rules of type $E/b^r \rightarrow b'$, where $b, b' \in \{a, \overline{a}\}$, are the spiking rules and they are used only if the neuron contains n bs such that $b^n \in L(E)$ and $n \geq r$. When neuron σ_i sends b' (a spike/anti-spike), it is replicated in such a

way that a spike/anti-spike is sent to all neurons σ_k such that $(i,k) \in syn$.

The rules of type $b^s \to \lambda$ are the forgetting rules; s spikes/anti-spikes are simply removed ("forgotten") when applying the rule. Like in the case of spiking rules, the left hand side of a forgetting rule must "cover" the contents of the neuron, that is, $a^s \to \lambda$ is applied only if the neuron contains exactly s spikes.

A spike/anti-spike emitted by neuron σ_i will pass immediately to all neurons σ_k such that $(i,k) \in syn$. That means transmission of spikes/anti-spikes takes no time, the spikes/anti-spikes will be available in neuron σ_k in the next step. There is an additional fact that a and $\overline{a}$ cannot stay together, they annihilate each other. If a neuron has either objects a or objects $\overline{a}$, and further objects of either type (maybe both) arrive from other neurons, such that we end with a^r and $\overline{a}^s$ inside, then immediately an annihilation rule $a\overline{a} \to \lambda$ (which is implicit in each neuron), is applied in a maximal manner, so that either a^{r-s} or $(\overline{a})^{s-r}$ remain for the next step, provided that $r \geq s$ or $s \geq r$, respectively. This mutual annihilation of spikes and anti-spikes takes no time and the annihilation rule has priority over spiking and forgetting rules, so each neuron always contains either only spikes or anti-spikes. Like in [79], we avoid using rules $\overline{a}^c \to \overline{a}$, but not the other three types, corresponding to the pairs (a,a), $(a,\overline{a})$, $(\overline{a},a)$. If we have a rule $E/b^r \to b'$ with $L(E) = \{b^r\}$, then we write it in the simplified form $b^r \to b'$.

Definition 5.1 (**Configuration**). *The configuration of the system is described by* $C = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ *where* $|\alpha_i|$ *is the number of spikes/anti-spikes present in the neuron* σ_i. $\alpha_i < 0$ *denotes that neuron* σ_i *is having* $|\alpha_i|$ *anti-spikes and* $\alpha_i > 0$ *represents that neuron* σ_i *is having* α_i *spikes. With this notation, the initial configuration of the system is described by* $C_0 = \langle n_1, n_2, \ldots, n_m \rangle$.

The SN PA system works in the same manner as the standard SN P system. A global clock is assumed and in each time unit, each neuron which can use a rule should do it (the system is synchronized), but the work of the system is sequential locally: only (at most) one rule is used in each neuron except the annihilation rule

which fires maximally with highest priority. If a neuron σ_i has more than one rule enabled, then only one of them is chosen non-deterministically. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other. In each step, all neurons which can use a rule of any type, spiking or forgetting, have to evolve, using a rule.

Definition 5.2 (**Vector Rule**)**.** *A vector rule of* Π *is a tuple* $\boldsymbol{v} \stackrel{df}{=} \langle \boldsymbol{1j}_1, \boldsymbol{2j}_2, \ldots, \boldsymbol{mj}_m \rangle$ *where, for each neuron* σ_i, $\boldsymbol{ij}_i$ *is either* ***i0*** *(when no rule is enabled) or an enabled rule* $\boldsymbol{ij}_i$ *from* R_i.

If a vector rule $\boldsymbol{v}$ is enabled at a configuration $\mathcal{C}$=$\langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ then $\mathcal{C}$ can evolve to $\mathcal{C}'$ =$\langle \alpha'_1, \alpha'_2, \ldots, \alpha'_m \rangle$, where
$\alpha'_i = \alpha_i - lhs(\boldsymbol{v}(i)) + \sum_{(k,i)\in syn} rhs(\boldsymbol{v}(k))$, $1 \leq i \leq m$
where $|lhs(\boldsymbol{v}(i))|$ and $|rhs(\boldsymbol{v}(i))|$ gives the number of spikes/anti-spikes consumed and sent by the rule $\boldsymbol{v}(i)$ respectively. $lhs(\boldsymbol{v}(i)) \stackrel{df}{=} -m_i$ if $\boldsymbol{v}(i)$ consumes m_i anti-spikes and $lhs(\boldsymbol{v}(i)) \stackrel{df}{=} m_i$ if $\boldsymbol{v}(i)$ consumes m_i spikes. Similarly $rhs(\boldsymbol{v}(i)) \stackrel{df}{=} 1$ if $\boldsymbol{v}(i)$ sends a spike and $rhs(\boldsymbol{v}(i)) \stackrel{df}{=} -1$ if $\boldsymbol{v}(i)$ sends an anti- spike. We can observe that annihilation rule is automatically applied since the spikes are represented using positive numbers and anti-spikes using negative numbers.

Definition 5.3 (**Transition**)**.** *Using the vector rule, we pass from one configuration of the system to another configuration, such a step is called a transition. For two configurations* $\mathcal{C}$ *and* $\mathcal{C}'$ *of* Π *we denote by* $\mathcal{C} \stackrel{\boldsymbol{v}}{\Longrightarrow} C'$*, if there is a direct transition from* $\mathcal{C}$ *to* $\mathcal{C}'$ *on the vector rule* $\boldsymbol{v}$.

A computation of Π is a finite or infinite sequences of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called reachable. A computation halts if it reaches a configuration where no rule can be used. In the generative mode, one of the neuron is considered as the output neuron and it sends output to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1, the moments of time when an

anti-spike emitted is marked with 0 and no output moments are just ignored. This binary sequence is called the spike train of the system - it might be infinite if the computation does not stop. With halting configurations, we associate a language, the binary strings describing the spike trains.

Let $\gamma = \mathcal{C}_0 \stackrel{\boldsymbol{v}_1}{\Longrightarrow} \mathcal{C}_1 \stackrel{\boldsymbol{v}_2}{\Longrightarrow} \ldots \stackrel{\boldsymbol{v}_k}{\Longrightarrow} \mathcal{C}_k$ be an halting computation ($\mathcal{C}_0$ is the initial configuration, and $\mathcal{C}_{i-1} \stackrel{\boldsymbol{v}_i}{\Longrightarrow} \mathcal{C}_i$ is the ith transition of γ). Let us denote by $bin(\gamma)$ the string $b_1 b_2 \ldots b_k$ where $b_i \in \{0,1\}$ and $b_i = 1$ iff the output neuron of the system Π sends a spike into the environment in the step i of γ (i.e. $rhs(\boldsymbol{v}_i(i_0)) = 1$), $b_i = 0$ iff it sends an anti-spike (i.e. $rhs(\boldsymbol{v}_i(i_0)) = -1$), and $b_i = \lambda$ if the step i generated no output. We denote by B the binary alphabet $\{0,1\}$ and by $COM(\Pi)$, the set of all halting computations of Π. Moreover, we define the language generated by the SN PA system Π by $L(\Pi) = \{bin(\gamma) \mid \gamma \in COM(\Pi)\}$.

5.3 Translating SN PA Systems into Petri Nets

In this section, we propose a formal method to translate SN PA systems into Petri nets suitable for simulation using any Petri net tool that supports parallel execution of transitions and guard functions. Here we consider same Petri net variant which was defined in the previous chapter.

Three places are used to represent each neuron. The marking of the places p_{2i-1} and p_{2i} gives the number of spikes and anti-spikes present in the neuron σ_i respectively. The place p_{is} (it is same as place p_{is} in the previous chapter) is added to allow at most one transition to fire from each input place corresponding to σ_i. p_{2m+1} and p_{2m+2} are the places corresponding to the environment and respectively gives the number of spikes and anti-spikes sent out by the output neuron. Every spiking or forgetting rule is one-to-one mapped to a transition in T. Regular expressions are translated into guard functions which further control the firing of transitions. The annihilation rule in each σ_i is implemented using two transitions t_{ia} and t_{ib}. A guard function is

associated with t_{ia} so that it is enabled only if number of tokens in place p_{2i-1} (spikes) is greater than or equal to the number of tokens in place p_{2i} (anti-spikes). t_{ia} clears the contents of place p_{2i} and keeps $\mathcal{M}(p_{2i-1}) - \mathcal{M}(p_{2i})$ tokens in place p_{2i-1}. Similarly the guard function associated with t_{ib} is enabled only if number of tokens in place p_{2i-1} is less the number of tokens in place p_{2i} and keeps $\mathcal{M}(p_{2i}) - \mathcal{M}(p_{2i-1})$ tokens in place p_{2i}.

The annihilation rule is applicable in each neuron of SN PA system only after the application of spiking rules (since forgetting rules does not add any spikes/anti-spikes). To simulate this behaviour of an SN PA system, a place p_{2m+3} with no tokens is introduced and an outgoing arc from each transition that corresponds to a spiking rule is connected to the place p_{2m+3}. The place p_{2m+3} gets a token for each transition corresponding to the spiking rule fired in the step. All transitions that corresponds to spiking rules are associated with a guard that enables the transition if place p_{2m+3} has no tokens. The transitions corresponding to the annihilation rules are applied only if any transitions corresponding to spiking rules are applied in the previous step. To implement this concept a guard function $\mathcal{M}(p_{2m+3}) > 0$ is added to each transition corresponding to the annihilation rule. The place p_{2m+3} is also connected to the transition t_0 which fires with the annihilated transitions to clear the contents of the place p_{2m+3} and thus allows spiking transitions to fire in the next step. In the construction described below, the SN PA system is considered for the translation after adding a synopsis $(i_0, m+1)$ to syn.

Definition 5.4 (**SN PA system to labelled Petri net**). *Let* Π=*(*O, σ_1, σ_2, σ_3, ..., σ_m , *syn* , i_0*) be an SN PA system, then the corresponding labelled Petri net* $\mathcal{K} \stackrel{df}{=} (V, \mathcal{NL}_\Pi, \zeta)$, $\mathcal{NL}_\Pi = (P, T, F, W, G, \mathcal{M}_0)$, *where*

1. $V = \{0, 1\}$ *is an alphabet.*
2. *The components of* $\mathcal{NL}_\Pi$ *are defined as*
   (a) $P \stackrel{df}{=} \{p_1, p_2, \ldots, p_{2m}, p_{2m+1}, p_{2m+2}, p_{2m+3}\}$ *is the set of places.*

(b) $T \stackrel{df}{=} T_1 \cup T_2 \cup \ldots T_m \cup \{t_0\}$ *where* T_i *is a set of transitions corresponding to each neuron* $\sigma_i, 1 \leq i \leq m$.

(c) for each rule $ij \in R_i$, T_i *contains a distinct transition* $t = t_{ij}$ *with the following connectivity:*

$W(p_{is}, t) = W(t, p_{is}) = W(t, p_{2m+3}) = 1$

if ij *is of the form* $E/a^r \rightarrow b$ *where* $b = a$ *or* $b = \overline{a}$ *or* $b = \lambda$ *then*

$G(t_{ij}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{2i-1}) \in \Psi(L(E))$ *and* $\mathcal{M}(p_{2m+3}) = 0)$ *then return* $true$ *else return* $false$

set $W(p_{2i-1}, t) = r$

for each synopsis $(i, k) \in syn$ *do*

if $b = a$ *then set* $W(t, p_{2k-1}) = 1$

if $b = \overline{a}$ *then set* $W(t, p_{2k}) = 1$

end for

else if ij *is of the form* $E/\overline{a}^r \rightarrow b'$ *where* $b' = a$ *or* $b' = \lambda$ *then*

$G(t_{ij}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{2i}) \in \Psi(L(E))$ *and* $\mathcal{M}(p_{2m+3}) = 0)$ *then return* $true$ *else return* $false$

set $W(p_{2i}, t) = r$

for each synopsis $(i, k) \in syn$ *do*

if $b' = a$ *then set* $W(t, p_{2k-1}) = 1$

end for

end if

end for

set $W(p_{2m+3}, t_0) = \mathcal{M}(p_{2m+3})$.

(d) For each annihilation rule that is internally present in neuron σ_i, T_i *contains two transitions* t_{ia} *and* t_{ib}.

for $i = 1$ *to* m *do*

set $W(p_{2i-1}, t_{ia}) = \mathcal{M}(p_{2i})$, $W(p_{2i}, t_{ia}) = \mathcal{M}(p_{2i})$,

$W(p_{is}, t_{ia}) = 1$, *and* $W(t_{ia}, p_{is}) = 1$

$W(p_{2i-1}, t_{ib}) = \mathcal{M}(p_{2i-1}), W(p_{2i}, t_{ib}) = \mathcal{M}(p_{2i-1})$,

$W(p_{is}, t_{ib}) = 1$, *and* $W(t_{ib}, p_{is}) = 1$
$G(t_{ia}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{2i-1}) >= \mathcal{M}(p_{2i})$ *and* $\mathcal{M}(p_{2m+3}) > 0)$ *then return* $true$ *else return* $false$
$G(t_{ib}) \stackrel{df}{=}$ *if* $(\mathcal{M}(p_{2i}) > \mathcal{M}(p_{2i-1})$ *and* $\mathcal{M}(p_{2m+3}) > 0)$ *then return* $true$ *else return* $false$
end for
The execution of t_{ia} *consumes* $\mathcal{M}(p_{2i})$ *tokens from its input places and leaves* $\mathcal{M}(p_{2i-1}) - \mathcal{M}(p_{2i})$ *tokens in place* p_{2i-1}. *Similarly the execution of* t_{ib} *consumes* $\mathcal{M}(p_{2i-1})$ *tokens from its input places and leaves* $\mathcal{M}(p_{2i}) - \mathcal{M}(p_{2i-1})$ *tokens in place* p_{2i}.

(e) for $i = 1$ *to* m, *set*
$\mathcal{M}_0(p_{2i-1}) \stackrel{df}{=} n_i$ *if* $n_i > 0$
$\mathcal{M}_0(p_{2i}) \stackrel{df}{=} n_i$ *if* $n_i < 0$
$\mathcal{M}_0(p_{is}) \stackrel{df}{=} 1$

3. $\zeta : 2^T/\{\emptyset\} \rightarrow V$ *where* $\zeta(U) = 1$ *if* $\exists t \in U$ *such that* $W(t, p_{2m+1}) = 1$, $\zeta(U) = 0$ *if* $\exists t \in U$ *such that* $W(t, p_{2m+2}) = 1$, *and* $\zeta(U) = \lambda$ *otherwise.*

In order to prove the equivalence between SN PA systems and Petri nets, we equate the languages generated by both the systems.

To capture a very tight correspondence between the SN PA system Π and the corresponding Petri net $\mathcal{NL}_\Pi$, we introduce a straightforward bijection between configurations of Π and the configuration mapped sub markings of $\mathcal{NL}_\Pi$, based on the correspondence between places and neurons.

Let $\mathcal{C} = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle$ be a configuration of the SN PA system Π. The corresponding configuration mapped sub marking $\phi(\mathcal{C})$ of $\mathcal{NL}_\Pi$ is defined as $\phi(\mathcal{C}) \stackrel{df}{=} \langle \beta_1, \beta_2, \ldots, \beta_m \rangle$, where for $1 \leq i \leq m$,
$\phi(\mathcal{C})(\beta_i) \stackrel{df}{=} \mathcal{M}(p_{2i-1}) - \mathcal{M}(p_{2i})$
Similarly, for any vector rule $\boldsymbol{v} = \langle \boldsymbol{1j}_1, \boldsymbol{2j}_2, \ldots, \boldsymbol{mj}_m \rangle$ of Π enabled at configuration $\mathcal{C}$,

we define an enabled maximal step $\xi(\boldsymbol{v})$ of transitions of $\mathcal{NL}_\Pi$ such that $\xi(\boldsymbol{v}) \stackrel{df}{=} \{t_{ij_i} \mid \boldsymbol{v}(i) = \boldsymbol{ij}_i$ with $j_i \geq 1, 1 \leq i \leq m\}$. It is clear that ϕ is a bijection from the configurations of Π to the configuration mapped sub markings of $\mathcal{NL}_\Pi$, and that ξ is a bijection from vector rules of Π to enabled maximal steps of $\mathcal{NL}_\Pi$.

We now can formulate a fundamental property concerning the relationship between the dynamics of the SN PA system Π and that of the corresponding Petri net:
$\mathcal{C} \stackrel{\boldsymbol{v}}{\Longrightarrow} \mathcal{C}'$ *if and only if* $\phi(\mathcal{C})[\xi(\boldsymbol{v})\rangle_m \mathcal{M}_1[H\rangle_m \phi(\mathcal{C}')$.
where $\mathcal{M}_1$ is the intermediate configuration mapped sub marking of the Petri net between $\phi(\mathcal{C})$ and $\phi(\mathcal{C}')$ and H is an intermediate step of transitions.

Since the initial configuration of Π corresponds through ϕ to the initial sub marking of $\mathcal{NL}_\Pi$, the above immediately implies that the computations of Π coincide with the locally sequential and globally maximal concurrency semantics of the net $\mathcal{NL}_\Pi$.

It can be observed that the structure of neurons in Π is used in the definitions of the structure of the net $\mathcal{NL}_\Pi$ (i.e., in the definitions of places, transitions and the guard function). Let $\mathcal{C}$ be a configuration of Π and there is a vector rule $\boldsymbol{v}$ enabled at $\mathcal{C}$ reaching a configuration $\mathcal{C}'$. As there is a mapping between configuration and markings, $\phi(\mathcal{C})$ is the marking of net $\mathcal{NL}_\Pi$ corresponding to the configuration $\mathcal{C}$ of Π. There is a one-to-one mapping between the rules in the SN PA system and transitions in net. So there exists a maximal step $[\xi(\boldsymbol{v})\rangle$ enabled at the marking $\phi(\mathcal{C})$. After the execution of the step $[\xi(\boldsymbol{v})\rangle$ the Petri net reaches the marking $\mathcal{M}_1$ where the tokens in the place p_{2i-1} and p_{2i} gives the number of spikes and anti-spikes present in neuron σ_i before the application of annihilation rule. In order to implement the annihilation rule, we introduce a maximal step $H \stackrel{df}{=} \{t_{ia} \mid \mathcal{M}(p_{2i-1}) > \mathcal{M}(p_{2i}), 1 \leq i \leq m\} \cup \{t_{ib} \mid \mathcal{M}(p_{2i}) > \mathcal{M}(p_{2i-1}), 1 \leq i \leq m\} \cup \{t_0\}$ enabled at $\mathcal{M}_1$. After the execution of the step H, the system reaches the configuration $\phi(\mathcal{C}')$. So here we map each vector rule of the SN PA system with two consecutive maximal steps. We can prove

only if part in the similar way. So the evolution of the Petri net $\mathcal{NL}_\Pi$ is same as the evolution of the SN PA system Π.

We now extend the statement for sequences of transitions and sequences of steps.
$\gamma = \mathcal{C}_0 \stackrel{\boldsymbol{v}_1}{\Rightarrow} \mathcal{C}_1 \stackrel{\boldsymbol{v}_2}{\Rightarrow} \ldots \stackrel{\boldsymbol{v}_k}{\Rightarrow} \mathcal{C}_k$ is an halting computation of Π ***if and only if*** $\Im(\gamma) = \phi(\mathcal{C}_0)[\xi(\boldsymbol{v}_1)\rangle_m \mathcal{M}_1[H_1\rangle_m \phi(\mathcal{C}_1)[\xi(\boldsymbol{v}_2)\rangle_m \mathcal{M}_2[H_2\rangle_m \ldots [\xi(\boldsymbol{v}_k)\rangle_m \mathcal{M}_k[H_k\rangle_m \phi(\mathcal{C}_k)$ is the halting maximal step sequence of $\mathcal{NL}_\Pi$.

So the evolution of the Petri net $\mathcal{NL}_\Pi$ is same as the evolution of the SN PA system Π. That means $\gamma \in COM(\Pi)$ iff $\Im(\gamma) \in S_m(\mathcal{NL}_\Pi)$.

Let $\mathcal{C}_{i-1} \stackrel{\boldsymbol{v}_i}{\Rightarrow} \mathcal{C}_i$ is the ith step of γ and if $bin(v_i) = 1$. By the definition of bin, $bin(v_i) = 1$ iff $\boldsymbol{v}_i(i_0)$ is a spiking rule with $rhs(\boldsymbol{v}_i(i_0)) = 1$. From the construction of Petri net and the definition of $\xi(\boldsymbol{v}_i)$ and H_i, we observe that the step $\xi(\boldsymbol{v}_i)$ contains a transitions t with $W(t, p_{2m+1}) = 1$, which implies that $\zeta(\xi(\boldsymbol{v}_i)) = 1$ and H_i contains transitions with no outgoings arcs to places p_{2m+1} and p_{2m+2}. So by the definition of ζ, $\zeta(H_i) = \lambda$. The output generated the net after firing of steps $\xi(\boldsymbol{v}_i)$ and H_i is 1. Similarly we can prove that $bin(\boldsymbol{v}_i) = 0$ iff $\zeta(\xi(\boldsymbol{v}_i))\zeta(H_i) = 0\lambda = 0$. We extend this to the words generated by both systems. If $w = bin(\gamma) \in \{0,1\}^*$ iff $w = \zeta(\Im(\gamma))$.

From the above statement, we prove that $L(\Pi) = L^m(\mathcal{NL}_\Pi)$.

5.4 An Example

Consider the graphical representation of an SN P system with anti-spikes in Figure 5.1(a). It is formally denoted as
Π_3=($\{a, \overline{a}\}$, σ_1, σ_2, σ_3, σ_4, ***syn*** , 4), with
σ_1 = (3, $\{a^3/a \to a, a^3 \to \overline{a}\}$),
σ_2 = (1, $\{a \to a\}$),
σ_3 = (1, $\{a \to a\}$),

σ_4 = (1, $\{a \to \overline{a}\,, \overline{a} \to a\}$),
syn={(1, 2), (2, 1), (1,4), (4,1), (1,3), (3,1)}.

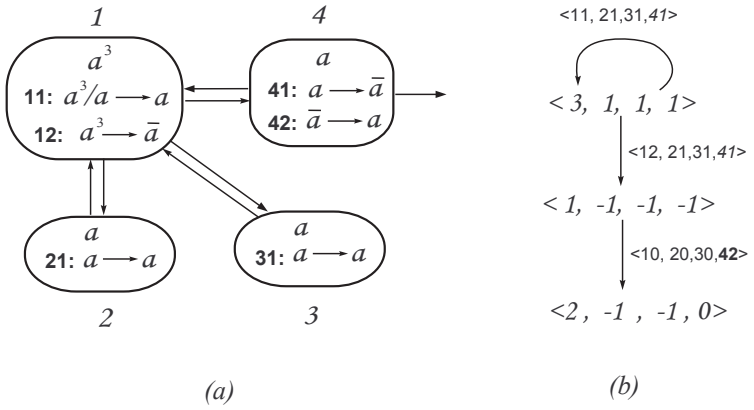


Figure 5.1: (a) An SN PA system Π_3 (b) Evolution of Π_3

The initial configuration of the system is $< 3,1,1,1 >$. The evolution of the system Π_3 can be analysed on a transition diagram as that from Figure 5.1(b).

Neuron σ_1 can behave non-deterministically choosing one of the two rules. As long as neuron σ_1 uses the rule $a^3/a \to a$, the computation cycles in the initial configuration sending a spike to neurons σ_2, σ_3 and σ_4; neuron σ_4 uses its first rule and sends an anti-spike to the environment and σ_1. Neurons σ_2 and σ_3 use their rules and send a spike to σ_1. So neuron σ_1 receives one anti-spike and two spikes (and two spikes are already present in it), after using annihilation rule, it will have three spikes.

If σ_1 uses its second rule $a^3 \to \overline{a}$, the three spikes are consumed and an anti-spike is sent to other three neurons. So σ_1 will have one spike and neurons σ_2, σ_3 and σ_4 will have one anti-spike each, reaching the configuration $< 1,-1,-1,-1 >$. In the next step neurons σ_1, σ_2 and σ_3 cannot fire and σ_4 uses the rule $\overline{a} \to a$ sending a spike to the environment and σ_1, reaching the configuration $< 2,-1,-1,0 >$ where the system halts. As the output neuron σ_4 is having a spike in its initial configuration it outputs at least one anti-spike (0), even if the σ_1 uses its second rule in the first

step.

Similar to SN P system, the transition diagram of a finite SN PA system can be interpreted as the representation of a non-deterministic finite automaton, with $\mathcal{C}_0$ being the initial state, the halting configurations being final states and each arrow being marked with 0 if in that transition the output neuron sends an anti-spike and with 1 if it sends a spike. In this way, we can identify the language generated by the system. In the case of finite SN PA system Π_3, the language generated is 0^+1.

Figure 5.2 shows the Petri net model $\mathcal{NL}_{\Pi_3}$ for the SN PA system Π_3 modelled using PNetLab. Each transition t_{ij} is named as $tl-tij$, where tl is the transition name given by the tool and tij is the transition name given as per methodology discussed in this chapter. Each place p_i is named as pi.

$p1$ and p_2 are places corresponding to neuron σ_1 for storing spikes and anti-spikes respectively. Similarly places $p(2i-1)$ and $p(2i)$ correspond to the neuron $\sigma_i, 1 \leq i \leq m$. The contents of places $p9$ and $p10$ respectively shows number of spikes and anti-spikes sent to the environment by the output neuron. The synchronizing places p_{is}, $1 \leq i \leq m$ are not required in PNetLab as the tool allows only one transition to fire from each input place. The symbol (1) inside the place indicates the presence of a token in that place.

By the definition of the Petri net, the difference $\mathcal{M}(p1) - \mathcal{M}(p2)$ gives the configuration of the first neuron. If we consider the configuration mapped sub marking i.e. $\mathcal{M}(p1) - \mathcal{M}(p2)$ for the neuron σ_1, $\mathcal{M}(p3) - \mathcal{M}(p4)$ for neuron σ_2, $\mathcal{M}(p5) - \mathcal{M}(p6)$ for neuron σ_3 and $\mathcal{M}(p7) - \mathcal{M}(p8)$ for neuron σ_4, the initial marking is $< 3, 1, 1, 1 >$ which is similar to the initial configuration of the SN PA system in Figure 5.1. In the step 1, after the firing of transitions $t5-t21, t6-t31, t7-t11, t9-t41$ (corresponding to rules ***21***,***31***,***11***,***41*** of Π_3), the system reaches the same sub marking $< 3, 1, 1, 1 >$ with $\mathcal{M}(p1) = 4$ and $\mathcal{M}(p2) = 1$, which gives respectively the number of spikes and anti-spikes of neuron σ_1. The place $p11$ receives 4 tokens since four spiking transitions are fired in the step. The place $p10$ receives a token, which represents the emitting of

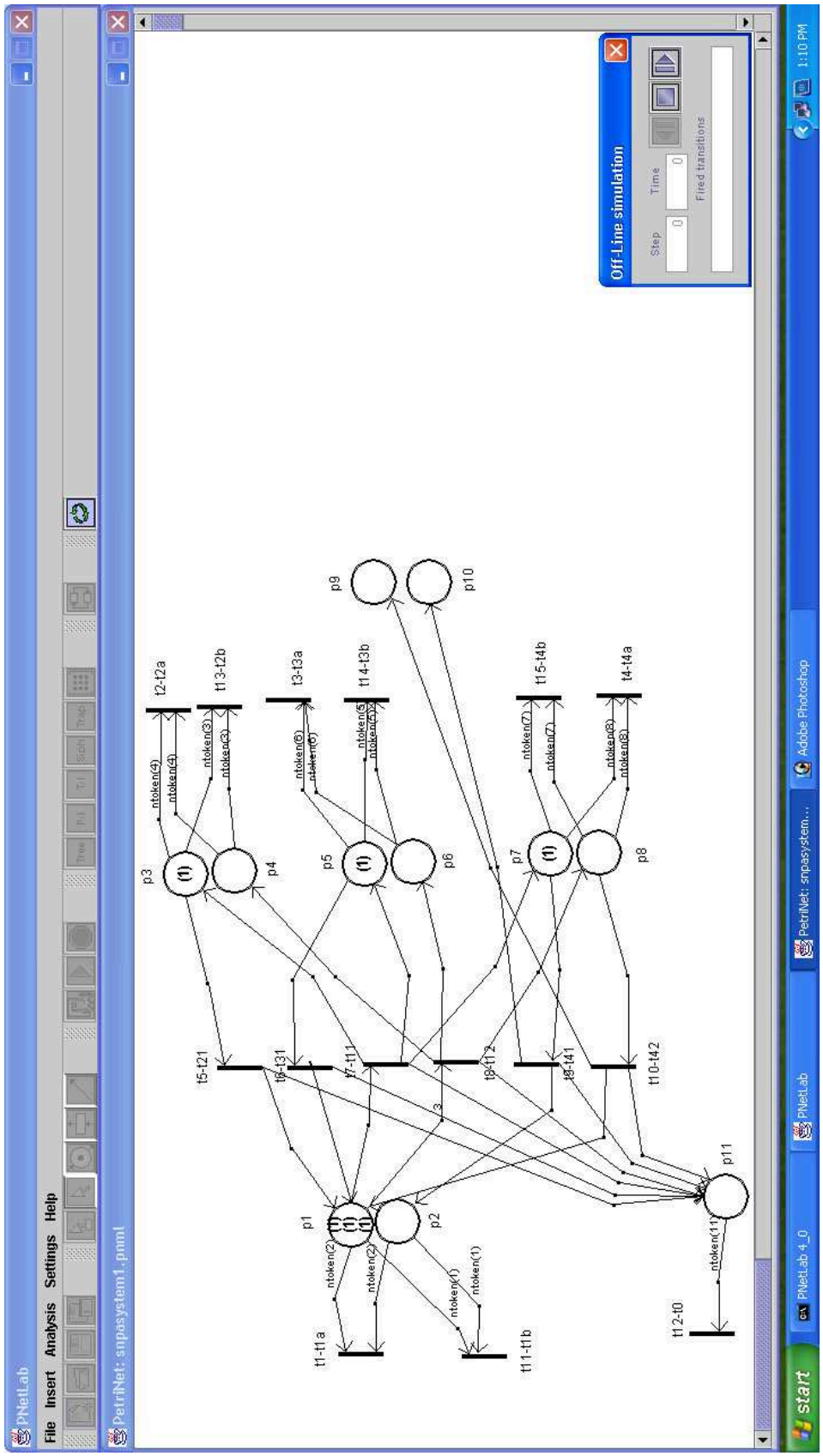
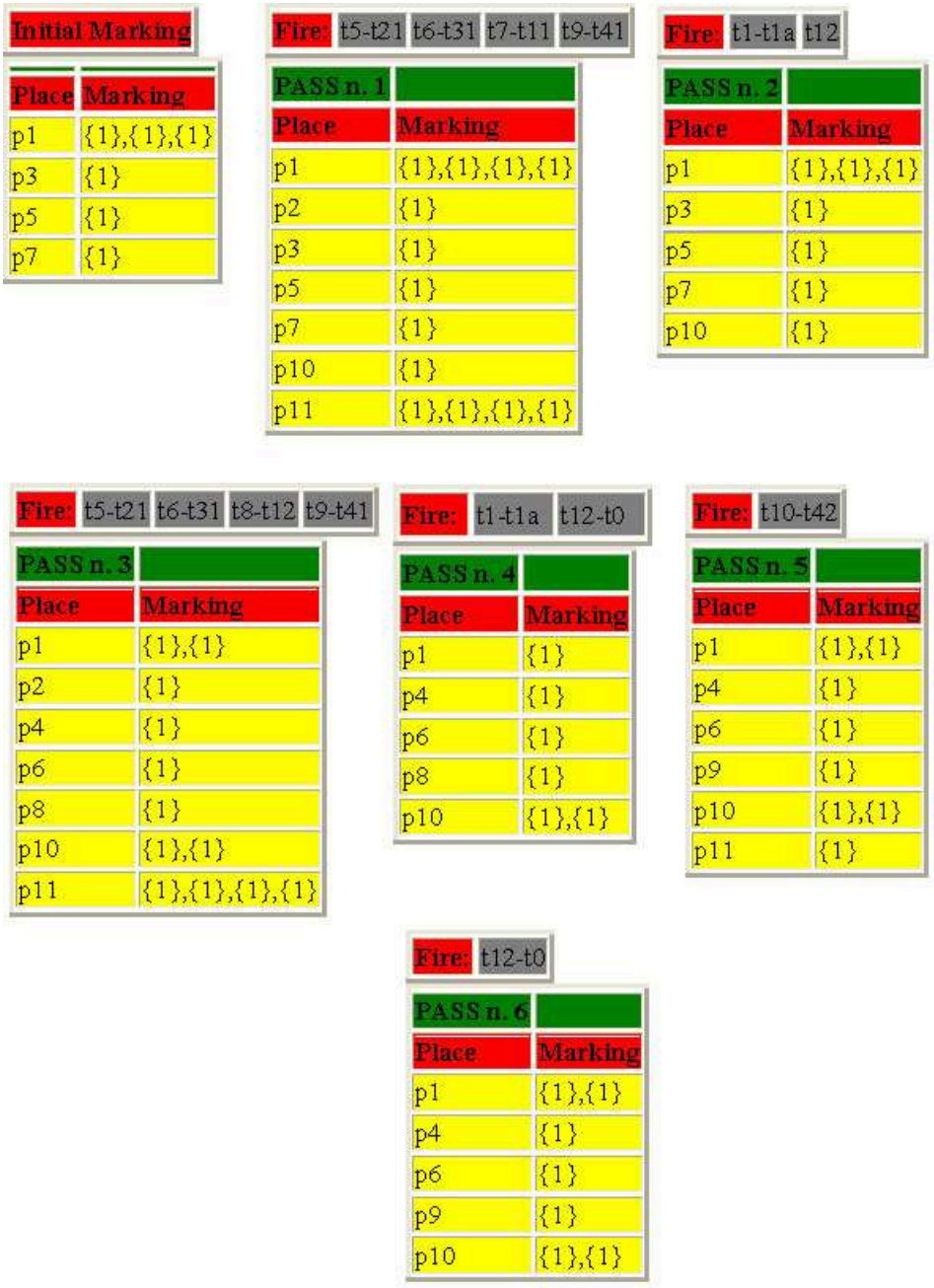


Figure 5.2: Petri net model for SN PA system Π_3



Initial Marking

Place	Marking
p1	{1},{1},{1}
p3	{1}
p5	{1}
p7	{1}

Fire: t5-t21 t6-t31 t7-t11 t9-t41

PASS n. 1

Place	Marking
p1	{1},{1},{1},{1}
p2	{1}
p3	{1}
p5	{1}
p7	{1}
p10	{1}
p11	{1},{1},{1},{1}

Fire: t1-t1a t12

PASS n. 2

Place	Marking
p1	{1},{1},{1}
p3	{1}
p5	{1}
p7	{1}
p10	{1}

Fire: t5-t21 t6-t31 t8-t12 t9-t41

PASS n. 3

Place	Marking
p1	{1},{1}
p2	{1}
p4	{1}
p6	{1}
p8	{1}
p10	{1},{1}
p11	{1},{1},{1},{1}

Fire: t1-t1a t12-t0

PASS n. 4

Place	Marking
p1	{1}
p4	{1}
p6	{1}
p8	{1}
p10	{1},{1}

Fire: t10-t42

PASS n. 5

Place	Marking
p1	{1},{1}
p4	{1}
p6	{1}
p9	{1}
p10	{1},{1}
p11	{1}

Fire: t12-t0

PASS n. 6

Place	Marking
p1	{1},{1}
p4	{1}
p6	{1}
p9	{1}
p10	{1},{1}

Figure 5.3: Report of markings of $\mathcal{NL}_{\Pi_3}$ in the steps of simulation in PNetLab

an anti-spike by the output neuron (shown in pass.1 of Figure 5.3). As the number of tokens in place $p11$ is greater than zero, the step 2 contains transitions corresponding to annihilation rules $t1 - t1a, t12 - t0$ and the system will be in the same sub marking but with $\mathcal{M}(p1) = 3$ and $\mathcal{M}(p2) = 0$ (shown in pass.2 of Figure 5.3). In the

step 3, the transitions $t5 - t21, t6 - t31, t8 - t12, t9 - t41$ are fired and the system reaches the sub marking $< 1, -1, -1, -1 >$ with $\mathcal{M}(p1) = 2$ and $\mathcal{M}(p2) = 1$ (shown in pass.3 of Figure 5.3). As the number of tokens in place $p11$ is 4, the transitions $t1 - t1a, t12 - t0$, corresponding to the annihilation rules will be fired in the step 4 again reaching the same configuration as that of SN PA system i.e. $< 1, -1, -1, -1 >$ (shown in pass.4 of Figure 5.3). The transitions enabled at this marking is $t10 - t42$ followed by the transitions corresponding to annihilation rules $t12 - t0$ reaching the final marking $< 2, -1, -1, 0 >$. Figure B.1$(a) - (f)$ in Appendix refapdx:b gives the output of the step-by-step simulation of the model in PNetLab. We can observe from the Figure 5.1 and Figure 5.3, that the configurations reachable from initial configuration of the SN PA system are same as the sub markings reachable in the corresponding Petri net model from the initial sub marking. So we conclude that the Petri net model in Figure 5.2 accurately simulates the working of the SN PA system Π_3.

5.5 Conclusion

SN PA systems are biologically inspired computing models that involve the use of two types of objects called spikes and anti-spikes and thus model the systems working with binary data in a very natural way. A formalism to study these models and validating them is needed. The Petri net tool called PNetLab allows the parallel execution of transitions. It enables to model the globally parallel firing semantics of all SN PA systems. They also allow the definition of functions on arcs and transitions. With numerous functionalities available with PNetLab, we succeeded in modelling the entire work of SN PA systems. At present the algorithms only enable the simulation of SN PA systems using P/T systems. It would be interesting to simulate SN PA systems using coloured Petri nets where we can use different colours for spikes and anti-spikes.

Chapter 6

Some Applications of SN P Systems

In this chapter, the relation between SN P systems and Petri nets is emphasized by focusing on modelling of producer/consumer paradigm and simplex stop-and-wait protocol. Here, we present SN P systems for producer/consumer problem and simplex stop-and-wait protocol. Then they are translated into Petri net models using the procedure proposed in Chapter 4. It is observed that there is a direct correspondence between the Petri net representation of the proposed models and standard solutions based on Petri nets already present in the literature.

6.1 Introduction

Petri nets are widely used for formal specification, analysis and verification of network protocols and distributed systems. The main reasons for this were that: (i) Petri nets allow one to describe these systems in a very adequate way (in particular, by directly supporting the fundamental notions of concurrency and asynchrony which are inherent to protocols and distributed systems); (ii) there exists a rich body of models, verification techniques and computer-aided tools based on Petri nets; and (iii) the visually appealing graphical interface makes Petri nets easy to understand

and manipulate for a wide range of practitioners.

As SN P systems are recently introduced distributed computational models and there is a similarity between the SN P systems and Petri nets, we therefore try to use SN P systems to model some of the systems that were simulated using Petri nets. We consider very simple systems like simplex stop-and-wait protocol with lossless communication channel and producer/consumer problem with buffer capacity one. Then, we translate our models of SN P systems into equivalent Petri nets with a corresponding semantics. The application of this construction to the producer/consumer problem and stop-and-wait protocol returns Petri nets representations which are same as the standard Petri nets solutions illustrated in [91] and [98] respectively.

In [11], the relationship between P systems and Petri nets was investigated by focusing on modelling producer/consumer problem with a buffer of capacity one item and a parallel version of this system where the producer and the consumer have direct access to two separate buffers, both of them having a capacity equal to one. Here we emphasize the relation between SN P systems and Petri nets by considering the problem of modelling producer/consumer problem with a buffer of capacity one item.

Let us consider the producer-consumer problem, in which two processes share a common buffer of capacity one item. One of them called a producer writes information into the buffer, and the other one, called consumer, reads and deletes it out. It is clear that, nothing can be written by the producer if the buffer is full. Similarly, nothing can be read and deleted by consumer if buffer is empty.

The simplex stop-and-wait is the simplest connection-less protocol for communication between two nodes. The system consists of a sender, receiver and communication channel. The protocol uses flow control with a window size of one, with the message sequence numbers simply alternate between 0 and 1.

The sender simply sends message packets numbered Pkt 0 or Pkt 1; the content of messages is not identified. These are acknowledged with Ack 0 or Ack 1 respectively. It solves the problem of congestion, as only one frame is outstanding at any time, frames cannot be lost due to congestion and the receiver will not be swamped by the sender. It assumes an error free communication channel. Messages are delivered immediately without loss. Here, messages are never lost and never have to be timed out and resent. It is easy to see that if a frame or an acknowledgement gets lost or damaged, a deadlock situation will occur where neither the sender nor the receiver can advance; they will be thrown into infinite loops. This is suitable for initial experimentation for representing it with SN P systems.

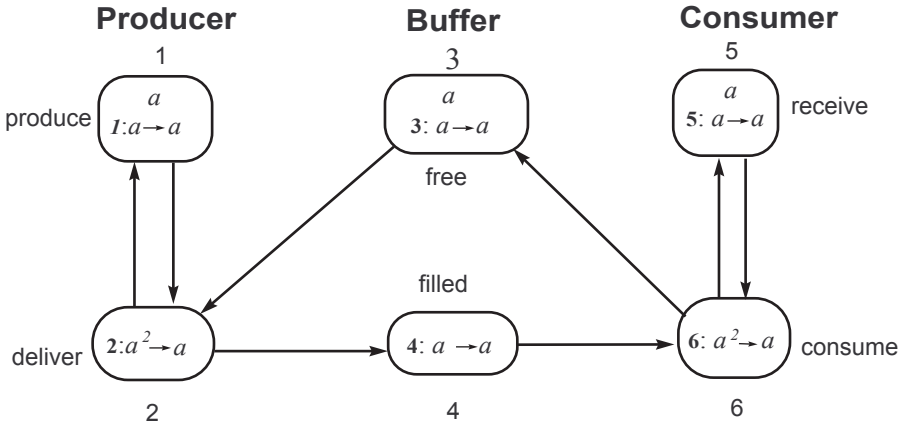


Figure 6.1: SN P system for producer/consumer problem

6.2 SN P System for Producer/Consumer Paradigm

We consider the problem of modelling producer/consumer systems: a system consisting of a producer and a consumer which synchronise through a buffer of capacity one item. Specifically, the producer has two states: “ready to produce” and “ready to deliver”; the consumer has two states, “ready to receive” and “ready to consume”;

the buffer is represented using two states: "filled" denoting the filled buffer and "free" denoting the free buffer. In state "ready to produce", the producer executes the operation "produce" and moves to state "ready to deliver"; in state "ready to deliver", if the buffer is free, the producer executes the operation "deliver", which fills the buffer cell, and moves back to state "ready to produce". Similarly, in state "ready to receive", if the buffer is full, the consumer executes the operation "receive", which empties the buffer, and moves to state "ready to consume"; in state "ready to consume", the consumer executes the operation "consume" and moves back to state "ready to receive".

The SN P system for the producer/consumer problem is described in Figure 6.1, with six neurons. The neurons σ_1 and σ_2 respectively represent "ready to produce" and "ready to deliver" states of the producer. Similarly neurons σ_3 and σ_4 represent "free" and "filled" states of the buffer respectively. The spike in σ_3 indicate that buffer is initially free. The neurons σ_5 and σ_6 respectively denote "ready to receive" and "ready to consume" states of the consumer. Initially neurons σ_1, σ_3 and σ_5 have spikes representing that producer is initially in "ready to produce" state, buffer is "free" and consumer is in "ready to receive" state. The neurons σ_1, σ_3 and σ_5 fire in the first step. The firing of the rule $a \to a$ in σ_1 represents the action "produce" and the producer transit from the state "ready to produce" to "ready to deliver" by sending a spike to σ_2. At the same time σ_3 also sends a spike to σ_2 indicating that buffer is empty and σ_5 sends spike to neuron σ_6 that it is ready to receive. In the second step, as the neuron σ_2 has two spikes it fires its rule and sends spikes to neurons σ_1 and σ_4 representing the "delivery action". The rule $a^2 \to a$ will be fired if the neuron σ_2 has two spikes, that is if the producer produced an item and the buffer is empty. After delivery the producer again comes to "ready to produce" state. The presence of a spike in neuron σ_4 indicates that buffer is full and fires using the rule $a \to a$. In the fourth step the consumer neuron σ_6 has two spikes and fires $a^2 \to a$ representing the action "consume" and sends spike to neuron σ_3 and σ_5 making the buffer cell "free" and consumer is back to "ready to receive" state and the cycle goes on.

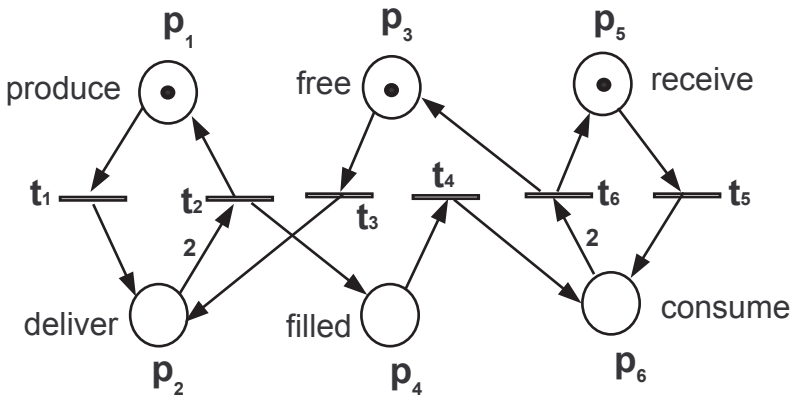


Figure 6.2: Petri net equivalent to SN P system in Figure 6.1

Petri net representation

We briefly recall the procedure to translate standard SN P system without delay Π=(O, σ_1, σ_2, ..., σ_m, *syn*, i_0) into corresponding Petri net $\mathcal{NL}_\Pi \stackrel{df}{=} (P, T, A, W, G, M_0)$ to Petri net (without labelling function). Every neuron in the SN P system is one-to-one mapped to a place in Petri net. Every rule is one-to-one mapped to a transition. Moreover, let us suppose the rules are labelled in a one-to-one manner with values in $\{1, 2, \ldots, k\}$, for some $k \geq 1$. Then the corresponding transition set $T = \{t_1, t_2, \ldots, t_k\}$. Regular expressions are translated into guard functions that further control the transitions. The bindings of transitions are found by matching incoming arc expressions with tokens marking input places and checking guard satisfaction. To describe locally sequential semantics of the SN P system, a synchronizing place is added to each place to allow at most one transition (one rule) to fire from each input place. Note that synchronizing place is not required during translation if the neuron is either having only one rule or no rule.

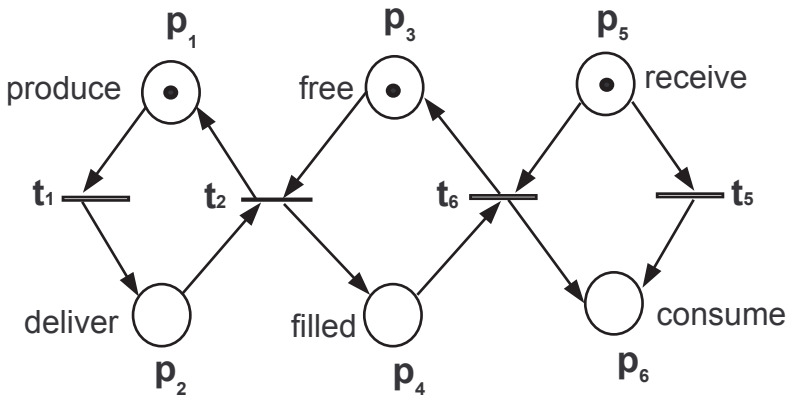


Figure 6.3: Reduced Petri net for producer/consumer problem

To capture the very tight correspondence between the SN P system without delay Π and Petri nets $\mathcal{NL}_{\Pi}$, we introduced a straight forward bijection between configurations of Π and markings of $\mathcal{NL}_{\Pi}$, based on the correspondence between places and neurons.

Using the above procedure, the SN P system for producer/consumer problem is translated into Petri net shown in Figure 6.2. The rules in neurons are one-to-one mapped to transitions and the corresponding transition set in the Petri net is $T = \{t_1, t_2, \ldots, t_6\}$. Each rule $\boldsymbol{i} : a^r \rightarrow a$ with $r \geq 1$ of neuron σ_i is mapped to a transition t_i with an incoming arc of weight r from the place p_i and outgoing arcs (t_i, p_j) of unit weight, where $(i, j) \in syn$ and $1 \leq j \leq m$. The transition t_i is assigned with a guard defined as $G(t_i) \stackrel{df}{=} \textit{if } (\mathcal{M}(p_i) = r) \textit{ then}$ return $true$ else return $false$ (marking $\mathcal{M}(p_i)$ gives the number of tokens in place p_i). As each neuron is having only one rule, there is only one outgoing arc from each place. We observe that for each place in the Petri net, the maximum number of tokens that can reside in the place is equal to the weight of its outgoing arc which in turn equal to the number of tokens needed in that place to enable its outgoing transition. So we can eliminate all the guard functions from the Petri net.

It is very important to develop methods of transformations which allow hierarchical or stepwise reductions and preserve the system properties to be analysed. In the literature, there are many reduction transformations in terms of places and transitions. Different approaches are discussed in [63, 65, 76, 97], where places and transitions are merged. Transition fusion is the most natural way to combine two or more transitions. To obtain the reduced equivalent Petri net, we here merge pair of transitions into a single transition.

In Figure 6.2, we observe that the firing of the transition t_2 requires two tokens in place p_2, one through t_1 and the other through t_3. The transition t_3 simply places a token from place p_3 into p_2. So the transition t_3 can be merged with t_2 by adding an input arc from place p_3 to t_2, removing the arc from t_3 to p_2 and replacing $W(p_2, t_2) = 2$ with $W(p_2, t_2) = 1$. Similarly the transition t_4 is merged with t_6. Hence, the Petri net is transformed into a new equivalent Petri net shown in Figure 6.3, which is same as the Petri net model of the producer/consumer system given in [91]. In other words, we observe a "direct" correspondence between the SN P system representation and the Petri net representation.

6.3 SN P System for Simplex Stop-and-Wait Protocol

We consider the problem of modelling simplex stop-and-wait protocol system. The stop and wait protocol is very easy to implement and runs very quickly and efficiently.

Specifically, the sender has two states: "wait Ack 0 (wait for acknowledgement of packet 0)" and "wait Ack 1". The receiver also has two states: "wait Pkt 0 (wait for packet 0)" and "wait Pkt 1". The channel has four states: "has Pkt 0", "has Ack 0", "has Pkt 1", "has Ack 1". In state "wait Ack 0", if the buffer "has ack 0", the sender executes the operation "send packet 1" and moves to state "wait Ack 1"; In state "wait Ack 1", if the buffer "has Ack 1", the sender executes the operation "send packet 0"

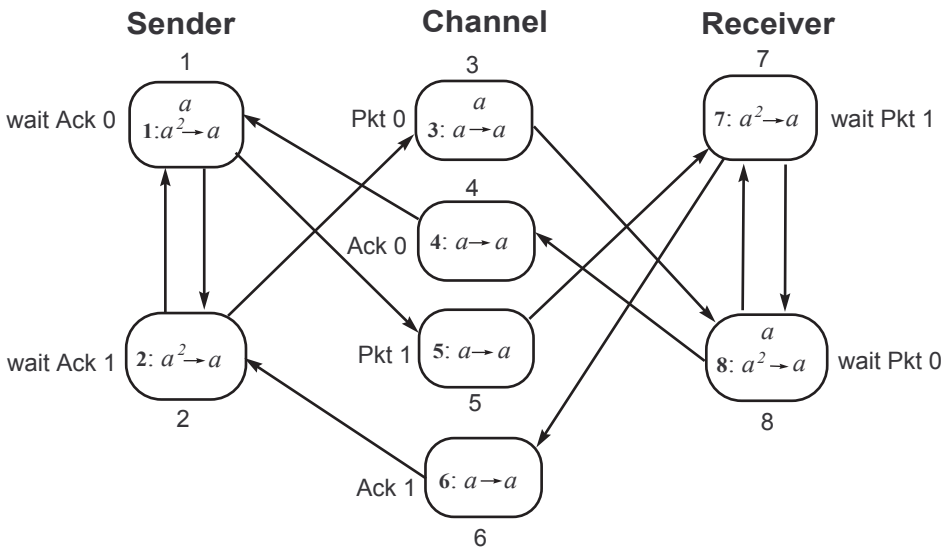


Figure 6.4: SN P system for simplex stop-and-wait protocol

and moves to state "wait Ack 0". Similarly, the receiver is in state "wait Pkt 0" and if the buffer "has Pkt 0", the receiver executes the operation "send Ack 0", and moves to state "wait Pkt 1"; In state "wait Pkt 1" and if the buffer "has Pkt 1", the receiver executes the operation "send Ack 1", and moves to state "wait Pkt 0".

In order to model this simplex stop-and-wait protocol with lossless communication channel, we consider an SN P system in Figure 6.4 with 8 neurons labelled in a one-to-one manner with values in σ_1 to σ_8. The neurons σ_1 and σ_2 respectively represent "wait Ack 0", "wait Ack 1" states of the sender. Similarly neurons σ_7 and σ_8 represent "wait Pkt 0" and "wait Pkt 1" states of the receiver respectively. The neurons σ_3, σ_4, σ_5 and σ_6 respectively represent the "has Pkt 0", "has Ack 0", "has Pkt 1" and "has Ack 1" states of the channel.

Initially neurons σ_1, σ_3, σ_8 have one spike each representing that sender is initially in "wait Ack 0" state, receiver is in "wait Pkt 0" state and channel "has Pkt 0". The neuron σ_3 fires in the first step using the rule $a \to a$ and sends a spike to σ_8 representing the action "deliver Pkt 0". In the next step σ_8 is having its required two

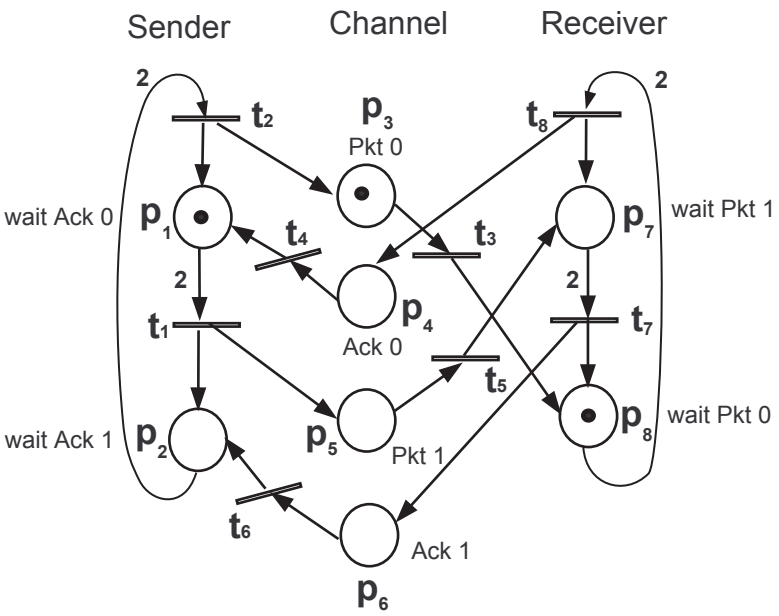


Figure 6.5: Petri net equivalent to SN P system in Figure 6.4

neurons, fires using its rule $a^2 \rightarrow a$ representing the action “send Ack 0” and sends one spike each to neurons σ_4 and σ_7 indicating that channel “has Ack 0” and receiver moves to the state “wait Pkt 1". In the third step the neuron σ_4 fires and sends its spike to neuron σ_1 representing the delivery action of the acknowledgement for packet 0. As neuron σ_1 has two spikes presenting that it has received the acknowledgement, it fires using the rule $a^2 \rightarrow a$ which represents the “send packet 1” operation. σ_1 sends one spike each to neurons σ_2 and σ_5. A spike in σ_2 indicates that sender is in “wait Ack 1” state and the channel “has Pkt 1”. The systems transmits the packet 1 in a similar way as the packet 0 and the cycle goes on.

Now let us consider the Petri net model of the simplex stop-and-wait protocol system given in [98] and it is reported in Figure 6.6. If we construct equivalent Petri net for the SN P system of Figure 6.4 using the procedure discussed in Chapter 4, we get a Petri net of Figure 6.5 which can be transformed to Petri net of Figure 6.6 by merging t_4 with t_1, t_6 with t_2, t_5 with t_7 and t_3 with t_8 in a similar way as we have done

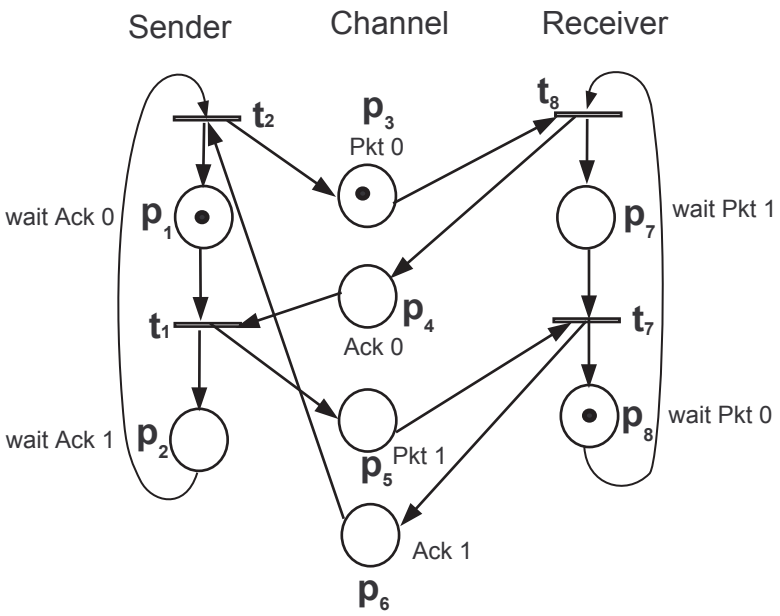


Figure 6.6: Reduced Petri net for simplex stop-and-wait protocol

in the case of producer/consumer problem.

6.4 Conclusion

As a part of building bridge between SN P systems and Petri nets, we try to model some systems using SN P systems that were modelled using Petri nets. We have constructed an SN P system for producer/consumer paradigm and simplex stop-and-wait protocol. Using the procedure described in this thesis, these systems are translated into equivalent Petri net with a corresponding semantics. We have observed that the solutions based on SN P systems proposed here are equivalent to the standard Petri nets representations.

Chapter 7

Conclusions and Future Work

Though spiking neural P systems is a new field of research motivated by the concept of spiking neurons, a lot of work has been done in this area from mathematical point of view. The systematic definition of SN P systems made them not only a robust, executable specification but also constitute the basic and essential starting point for deploying various methods and techniques to improve, formally verify or analyse these systems. In this work we concentrate on the way to simulate and study the behavioural properties of different variants of SN P systems.

Petri nets are one of the most widely used models of concurrency, which has attracted, since its introduction, the interest of both theoreticians and practitioners. Along the years Petri nets have been equipped with satisfactory semantics, justifying their intrinsically concurrent nature and which have served as basis for the development of a variety of modelling and verification techniques. We showed how the use of Petri nets for formal verification and analysis enrich our knowledge about the SN P systems. We now summarize the results presented in this thesis and describe some future work in this area.

We stated a series of investigations about spiking neural P system with anti-spikes. We studied the computational power of SN PA systems as language generators. Since they work with two objects (the spike and the anti spike), thus, provide the possibility of representing the generated strings in such a way that the non-firing steps of the output neuron are ignored, the firing of a spike generates the symbol 1, and the firing of an anti-spike generates the second symbol of the binary alphabet. The families of finite binary languages and regular binary languages are characterized. Furthermore, a characterization of recursively enumerable languages is given by using the SN PA systems. We compared the generative power of SN PA systems with that of SN P systems, demonstrating the increase of power obtained by the possibility of using anti-spikes. Spiking neural P systems with anti-spikes are also used as transducers. We gave an improved result of solving satisfiability problem in non-deterministic way using SN PA systems. We also showed that arithmetic operations can be performed on negative numbers by using SN PA systems.

In SN PA systems, if we label the non-firing steps of the output neuron also, we can get a spike train over a three letter alphabet: no output, producing spikes, and producing anti-spikes, respectively. This can be an interesting way to produce languages over three letters. It is worth investigating the languages generated over other alphabets with extended rules of the form $E/b^r \rightarrow b'^q$, where $b, b' \in \{a, \overline{a}\}$.

Here we considered Petri net as a tool to simulate and analyse some variants of SN P systems. We proposed a generalized procedure to translate standard SN P systems, extended SN P systems and SN P systems with anti-spikes into Petri net models. To compare SN P systems to the corresponding Petri nets, we related the languages generated by the systems. The procedure is illustrated with a series of examples.

We believe that the further use of a Petri net simulator would provide a very useful tool for checking the effectiveness of SN P systems and for returning meaningful information about its behavioural properties. Once the SN P systems have

been specified using Petri net formalism, certain tools can be used to study the properties. We propose the use of PNetLab - a Petri net tool for simulating the system in step-by-step mode.

Since PNetLab uses PNML interchange format to store the Petri net models, it would be interesting to use other tools on these Petri net models to study the behaviour properties of SN P systems.

Asynchronous and sequential SN P systems differ from extended SN P systems only in the mode of operation (i.e. operating in asynchronous or sequential modes). So we can use the same procedure to translate these systems into Petri net models. The obtained Petri nets models are then executed in sequential or asynchronous mode in order to study the behaviour of sequential or asynchronous SN P systems. Nevertheless, there are classes of SN P systems whose translations is yet to be investigated: SN P systems working in exhaustive mode, SN P systems with astrocyte like control, and SN P system with symport and antiport etc. These represent open problems which will be addressed by future research.

We also emphasized the relationship between spiking neural P systems and Petri nets by constructing SN P systems for *simplex stop-and-wait protocol* and *producer/consumer paradigm*. They are translated into equivalent Petri net models, which are observed as standard solutions based on Petri nets already present in the literature. As a part of research work one can try to simulate producer/consumer problem with buffer capacity more than one using SN P system. Similarly it can be investigated whether other network protocols can be simulated by SN P systems.

SN P systems contribute consistently to building a mature research area of membrane computing having strong links with computational models like formal verification and analysis. Hence, it is in our intentions to strongly develop this topic and motivate further cooperation between the areas of membrane computing and Petri nets.

Bibliography

[1] The P-Lingua web site:. *http://www.p-lingua.org/*.

[2] The P systems web page:. *http://ppage.psystems.eu/*.

[3] PNetLab: A Petri net tool. *http://www.automatica.unisa.it/PnetLab.html*.

[4] L. M. Adleman. Molecular computation of solutions on combinatorial problems. *Science*, 226:1021–1024, 1994.

[5] T. Agerwala and M. Flynn. Comments on capabilities, limitations and correctness of Petri nets. *Computer Architecture News*, 4(2):81–86, 1973.

[6] O. Agrigoroaiei and G. Ciobanu. Rewriting logic specification of membrane systems with promoters and inhibitors. *Electronic Notes Theoretical Computer Science*, 238(3):5–22, 2009.

[7] D. al Zilio and E. Formenti. On the dynamics of PB systems: a Petri net view. In Martín-Vide et al. [72], pages 153–167.

[8] A. Alhazov, R. Freund, M. Oswald, and M. Slavkovik. Extended spiking neural P systems. In Hoogeboom et al. [42], pages 123–134.

[9] A. Alhazov, R. Freund, M. Oswald, and M. Slavkovik. Extended variants of spiking neural P systems generating strings and vectors of non-negative integers. In H. J. Hoogeboom, Gh. Păun, and G. Rozenberg, editors, *Preproceedings of the Seventh International Workshop on Membrane Computing,Leiden, The Netherlands, July 17-21*, pages 88–101, 2006.

[10] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[11] F. Bernardini, M. Gheorghe, M. Margenstern, and S. Verlan. Producer/consumer in membrane systems and Petri nets. In S. B. Cooper, B. Löwe, and A. Sorbi, editors, *Computation and Logic in the Real World, Third Conference on Computability in Europe, CiE 2007, Siena, Italy, June 18-23, 2007, Proceedings*, volume 4497 of *Lecture Notes in Computer Science*, pages 43–52. Springer, 2007.

[12] A. Binder, R. Freund, M. Oswald, and L. Vock. Extended spiking neural p systems with excitatory and inhibitory astrocytes. In A. Aggarwal, R. Yager, and I. W. Sandberg, editors, *Proceedings of the 8th WSEAS International Conference on Evolutionary Computing (EC'07), Vancouver, Canada*, pages 320–325, June 18-20, 2007.

[13] H.-D. Burkhard. The maximum firing strategy in Petri nets gives more power. Technical Report 411, ICS-PAS, Warschau, 1980.

[14] H.-D. Burkhard. Ordered firing in Petri nets. *Journal of Information Processing and Cybernetics*, 17(2-3):71–86, 1981.

[15] F. Cabarle, H. Adorna, and M. A. Martínez-del-Amor. A spiking neural P system simulator based on CUDA. In Gheorghe et al. [38], pages 87–103.

[16] C. S. Calude, Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[17] M. Cavaliere, O. H. Ibarra, Gh. Păun, Ö. Egecioglu, M. Ionescu, and S. Woodworth. Asynchronous spiking neural P systems. *Theoretical Computer Science*, 410(24-25):2352–2364, 2009.

[18] H. Chen, R. Freund, M. Ionescu, Gh. Păun, and M. J. Pérez-Jiménez. On string languages generated by spiking neural P systems. *Fundamenta Informaticae*, 75(1-4):141–162, 2007.

[19] H. Chen, M. Ionescu, T. Ishdorj, A. Păun, Gh. Păun, and M. J. Pérez-Jiménez. Spiking neural P systems with extended rules: universality and languages. *Natural Computing*, 7(2):147–166, 2008.

[20] G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In R. Valette, editor, *Proceedings of 15th International Conference on*

Applications and Theory of Petri Nets, Zaragoza, Spain, June 1994, volume 815 of *Lecture Notes in Computer Science*, pages 179–198. Springer-Verlag, 1994.

[21] G. Ciobanu, Gh. Păun, and M. J. Pérez-Jiménez, editors. *Applications of Membrane Computing*. Natural Computing Series. Springer, 2006.

[22] D. Díz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua programming environment for membrane computing. In *Membrane Computing. 9th International Workshop, WMC 2008, Edinburgh, UK, July 28-31, 2008, Revised Selected and Invited Papers*, volume 5391 of *Lecture Notes in Computer Science*, pages 187–203. Springer, 2009.

[23] R. Freund, Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *Membrane Computing. 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*, volume 3850 of *Lecture Notes in Computer Science*. Springer, 2006.

[24] P. Frisco. About P systems with symport/antiport. *Soft Computing*, 9(9):664–672, 2005.

[25] P. Frisco. P systems, Petri nets and program machines. In Freund et al. [23], pages 209–223.

[26] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0. In Gh. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing. 10th International Workshop, WMC 2009, Curtea de Arges, Romania, August 24-27, 2009, Revised Selected and Invited Papers*, volume 5957 of *Lecture Notes in Computer Science*, pages 264–288. Springer, 2010.

[27] H. J. Genrich and K. Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[28] Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[29] Gh. Păun. P systems with active membranes: attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75–90, 2001.

[30] Gh. Păun. *Membrane Computing. An Introduction*. Springer, Berlin, 2002.

[31] Gh. Păun. Spiking neural P systems used as acceptors and transducers. In J. Jan Holub and J. Zdárek, editors, *Implementation and Application of Automata, 12th International Conference, CIAA 2007, Prague, Czech Republic, July 16-18, 2007, Revised Selected Papers*, volume 4783 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2007.

[32] Gh. Păun. Spiking neural P systems with astrocyte-like control. *Journal of Universal Computer Science*, 13(11):1701–1721, 2007.

[33] Gh. Păun. Twenty six research topics about spiking neural P systems. In M. A. Gutiérrez-Naranjo, Gh. Păun, A. Romero-Jiménez, and A. Núñez, editors, *Proceedings of the Fifth Brainstorming Week on Membrane Computing, Sevilla, Spain, January 29 - February 2*, pages 153–169, 2007.

[34] Gh. Păun, M. J. Pérez-Jiménez, and G. Rozenberg. Spike trains in spiking neural P systems. *International Journal of Foundations of Computer Science*, 17(4):975–1002, 2006.

[35] Gh. Păun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.

[36] Gh. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing. New Computing Paradigms*. Springer-Verlag, Berlin, 1998.

[37] Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.

[38] M. Gheorghe, Gh. Păun, G. Rozenberg, A. Salomaa, and S. verlan, editors. *Membrane Computing - 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers*, volume 7184 of *Lecture Notes in Computer Science*. Springer, 2012.

[39] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and D. Ramírez-Martínez. A software tool for verification of spiking neural P systems. *Natural Computing*, 7(4):485–497, 2008.

[40] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Available membrane computing software. In Ciobanu et al. [21], pages 411–439.

[41] M. Hack. Petri net languages. Technical Report MIT-LCS-TR-159, Massachusetts Institute of Technology, 1976.

[42] H. J. Hoogeboom, Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *Membrane Computing. 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised Selected and Invited Papers*, volume 4361 of *Lecture Notes in Computer Science*. Springer, 2006.

[43] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[44] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[45] O. H. Ibarra, M. J. Pérez-Jiménez, and T. Yokomori. On spiking neural P systems. *Natural Computing*, 9(2):475–491, 2010.

[46] O. H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosik, and S. Woodworth. Normal forms for spiking neural P systems. *Theoretical Computer Science*, 372(2-3):196–217, 2007.

[47] O. H. Ibarra and S. Woodworth. Characterizations of some restricted spiking neural P systems. In Hoogeboom et al. [42], pages 424–442.

[48] O. H. Ibarra and S. Woodworth. Characterizations of some classes of spiking neural P systems. *Natural Computing*, 7(4):499–517, 2008.

[49] O. H. Ibarra, S. Woodworth, H.-C. Yen, and Z. Dang. On the computational power of 1-deterministic and sequential P systems. *Fundamenta Informaticae*, 73(1-2):133–152, 2006.

[50] M. Ionescu, Gh. Păun, and T. Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2-3):279–308, 2006.

[51] M. Ionescu, Gh. Păun, and T. Yokomori. Spiking neural P systems with exhaustive use of rules. *International Journal of Unconventional Computing*, 3(2):135–153, 2007.

[52] M. Ionescu and D. Sburlan. Some applications of spiking neural P systems. *Computing and Informatics*, 27(3+):515–528, 2008.

[53] M. Jantzen and G. Zetzsche. Labelled step sequences in Petri nets. In K. M. van Hee and R. Valk, editors, *Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings*,

volume 5062 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2008.

[54] K. Jenson. *Coloured Petri nets: Basic Concepts, Analysis, Methods and Practical Use*. EACTS-Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[55] J. Kleijn and M. Koutny. Synchrony and asynchrony in membrane systems. In Hoogeboom et al. [42], pages 66–85.

[56] J. Kleijn and M. Koutny. Processes of Petri nets with range testing. *Fundamenta Informaticae*, 80:199–219, 2007.

[57] J. Kleijn and M. Koutny. Processes of membrane systems with promoters and inhibitors. *Theoretical Computer Science*, 404:112–126, 2008.

[58] J. Kleijn and M. Koutny. A Petri net model for membrane systems with dynamic structure. *Natural Computing*, 8(4):781–796, 2009.

[59] J. Kleijn, M. Koutny, and G. Rozenberg. Process semantics for membrane systems. *Journal of Automata, Languages and Combinatorics*, 11(3):321–340, 2006.

[60] J. Kleijn, M. Koutny, and G. Rozenberg. Towards a Petri net semantics for membrane systems. [23], pages 292–309.

[61] S. N. Krishna. P systems with symport/antiport: The traces of RBCs. In G. Mauri, Gh. Păun, M. J. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing. 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*, pages 331–343. Springer, 2005.

[62] M. Kudlek. Sequentiality and parallelity in Petri nets. In E. Kindler, editor, *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN-04)*, pages 55–60. Trier, Springer, September 30-October 1 2004.

[63] Y. S. Kwong. On reduction of asynchronous systems. *Theoretical Computer Science*, 5(1):25–50, 1977.

[64] L. J. Landau and J. G. Taylor, editors. *Concepts for Neural Networks: A Survey*. Springer-Verlag, New York, 1997.

[65] K. H. Lee and J. Favrel. Hierarchical reduction method for analysis and decomposition of Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(2):272–280, 1985.

[66] A. Leporati, G. Mauri, C. Zandron, Gh. Păun, and M. J. Pérez-Jiménez. Uniform solutions to SAT and subset-sum by spiking neural P systems. *Natural Computing*, 8(4):681–702, 2009.

[67] W. Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):39–43, 2002.

[68] L. F. Macías-Ramos, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia-Cabrera, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua based simulator for spiking neural P systems. In Gheorghe et al. [38], pages 257–281.

[69] M. Malita. Membrane computing in Prolog. In *Pre-Proceedings of the Multiset Workshop on Multiset Processing, Curtea de Arges, Romania, CDMTCS, University of Auckland*, pages 159–176, 21-25, August 2000.

[70] M. Margenstern and Y. Rogozhin, editors. *P Systems with Membrane Creation: Universality and Efficiency*, volume 2055 of *Lecture Notes in Computer Science*. Springer, 2001.

[71] C. Martín-Vide, Gh. Păun, J. Pazos, and A. Rodríguez Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[72] C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *Membrane Computing. International Workshop, WMC 2003, Tarragona, Spain, July 17-22, 2003, Revised Papers*, volume 2933 of *Lecture Notes in Computer Science*. Springer, 2004.

[73] A. G. Miguel and L. Alberto. First steps towards a CPU made of spiking neural P systems. *International Journal of Computers, Communications and Control*, 4:244–252, 2009.

[74] M. Minsky. *Computation Finite and infinite machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.

[75] M. K. Molloy. *On the integration of delay and throughput measures in distributed processing models*. PhD thesis, University of California, Los Angeles, 1981.

[76] T. Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[77] T. Neary. A boundary between universality and non-universality in extended spiking neural P systems. In A. H. Dediu, H. Fernau, and C. Martín-Vide, editors, *Proceedings of Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010*, volume 6031 of *Lecture Notes in Computer Science*, pages 475–487. Springer, 2010.

[78] T. Neary. A universal spiking neural P system with 11 neurons. In M. Marian Gheorghe, T. Hinze, and Gh. Păun, editors, *Proceedings of the Eleventh International Conference on Membrane Computing (CMC-11), Jena, Germany*, pages 65–74, 24-27 August 2010.

[79] L. Pan and Gh. Păun. Spiking neural P systems with anti-spikes. *International Journal of Computers, Communications and Control*, 4(3):273–282, 2009.

[80] L. Pan and Gh. Păun. Spiking neural P systems: An improved normal form. *Theoretical Computer Science*, 411(6):906–918, 2010.

[81] L. Pan, Gh. Păun, and M. J. Pérez-Jiménez. Spiking neural P systems with neuron division and budding. *SCIENCE CHINA Information Sciences*, 54(8):1596–1607, 2011.

[82] L. Pan and T.-O. Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, 10(5):630–649, 2004.

[83] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[84] J. L. Peterson. *Petri net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[85] C. A. Petri. *Kommunication mit Automaten*. PhD thesis, Institut fur Instrumentelle Mathematik, Schriften des IIM 2, Bonn, Germany, 1962.

[86] N. Pisanti. A survey on DNA computing. Technical Report TR-97-07, Dipartimento di Informatica, Universita Di Pisa, 1997.

[87] F. Pommereau. Petri nets as executable specifications of high-level timed parallel systems. Technical Report TR-2003-09, Laboratory of Algorithms, Complexity and Logic University of Paris XII, Val-de-Marne, 2004.

[88] A. Păun and Gh. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–306, 2002.

[89] Z. Qi, J. You, and H. Mao. P systems and Petri nets. In Martín-Vide et al. [72], pages 286–303.

[90] C. Ramchandani. *Performance evaluation of asynchronous concurrent systems by timed Petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, 1973.

[91] W. Reisig. *Elements of Distributed Algorithms. Modelling and Analysis with Petri Nets*. Springer, Heidelberg, 1998.

[92] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets*. Lecture Notes in Computer Science, 1491, 1492, Springer-Verlag, Berlin, 1998.

[93] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, Heidelberg, 1997.

[94] A. Salomaa. *Formal Languages*. Academic Press, 1973.

[95] J. Sifakis. Use of Petri nets for performance evaluation. In H. Beilner and E. Erol Gelenbe, editors, *Measuring, Modelling and Evaluating Computer Systems, Proceedings of the Third International Symposium, Bonn - Bad Godesberg, Germany*, pages 75–93, 3-5 October 1977.

[96] D. R. Simon. On the power of quantum computation. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA*. IEEE Computer Society Press, 20-22 November 1994.

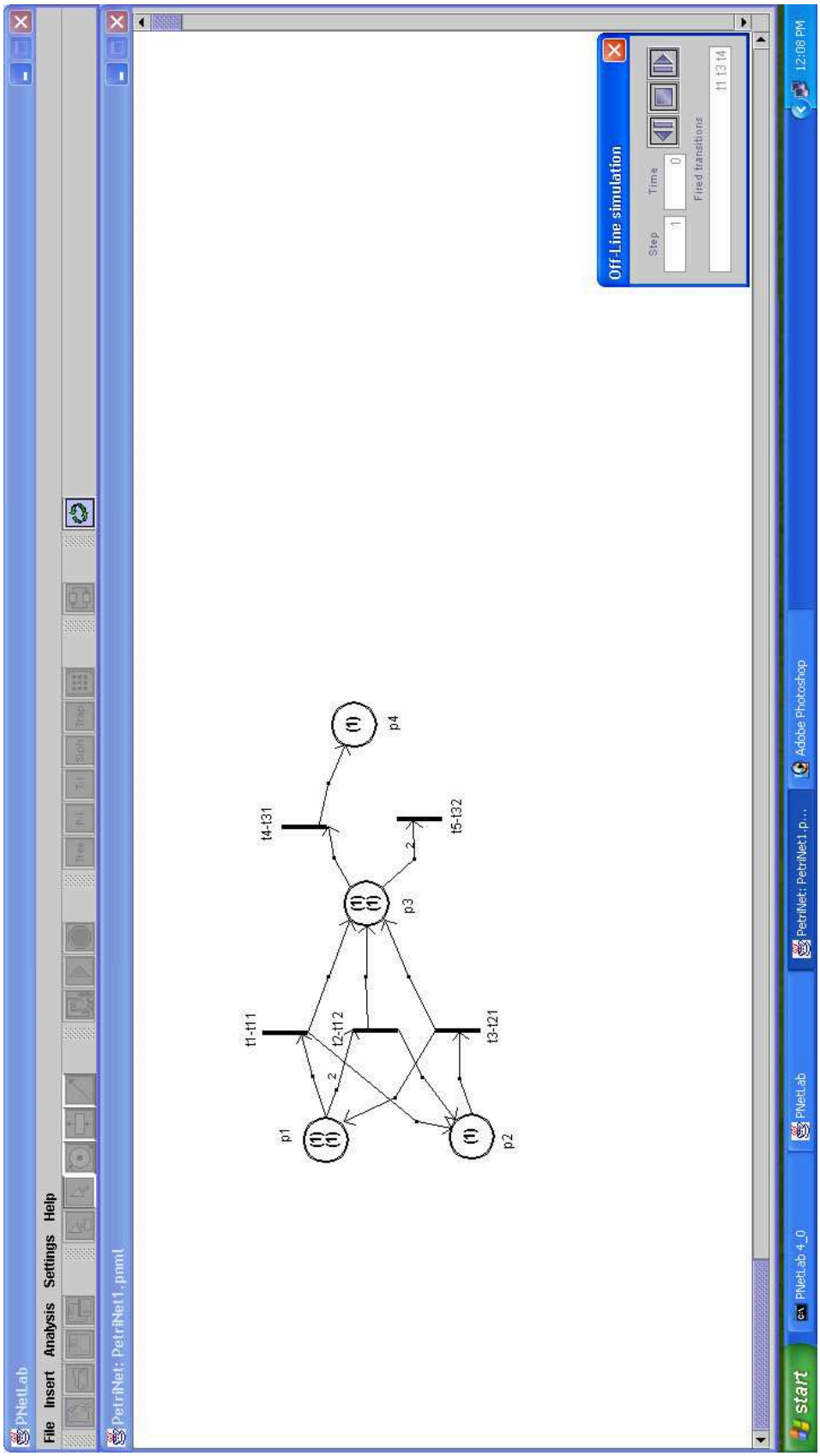
[97] I. Suzuki and T. Murata. A method for stepwise refinement and abstraction of Petri nets. *Journal of Computer and Systems Science*, 27(1):51–76, 1983.

[98] M. Yusufu. Petri nets. *http://www.cas.mcmaster./sartipi/course/cas707/w07/slides/Mar13-PetriNets-Munina.pdf*.

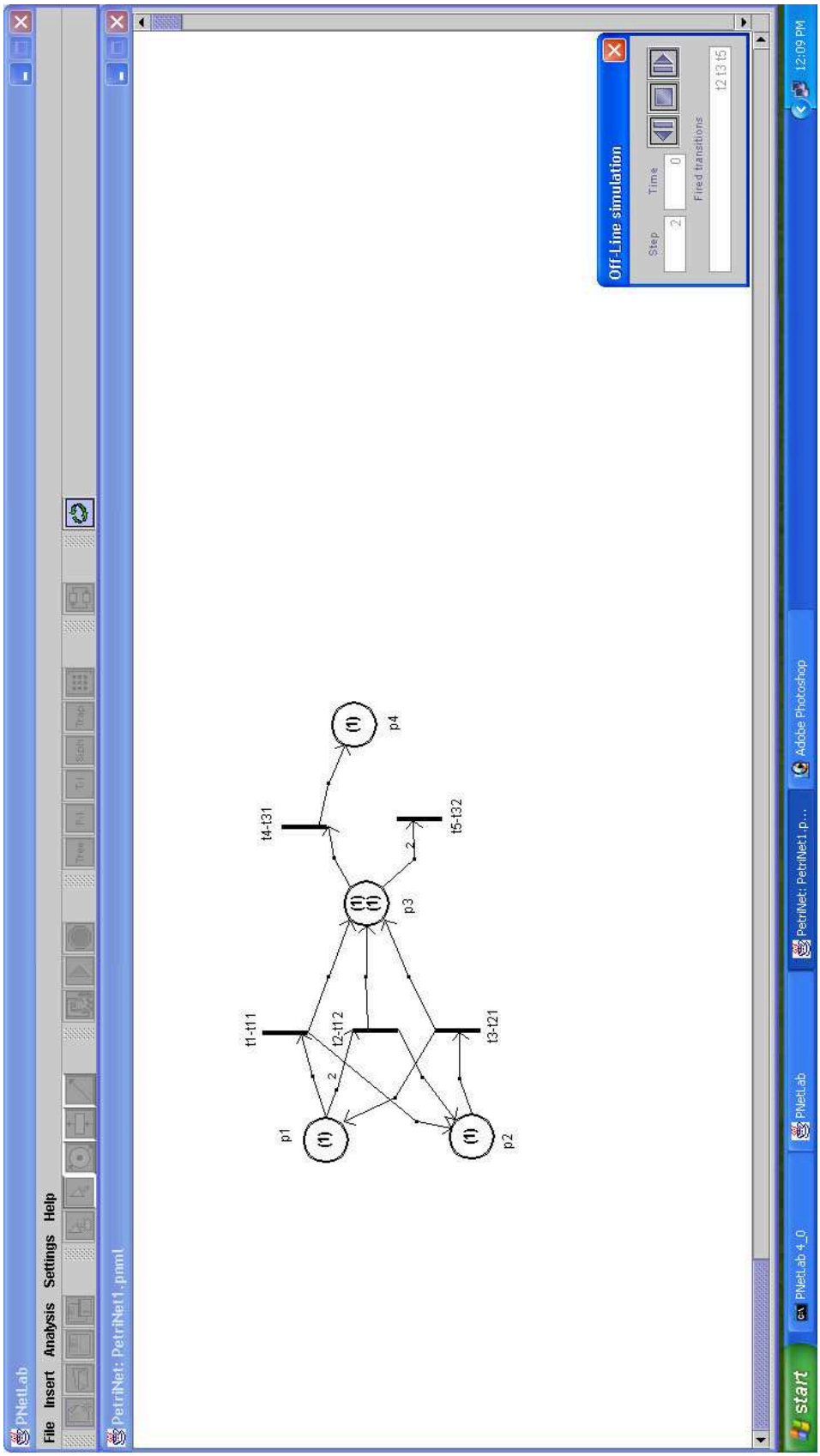
[99] X. Zeng, H. Adorna, M. A. Martínez-del Amor, L. Pan, and M. Pérez-Jiménez. Matrix representation of spiking neural P systems. In M. Gheorghe, T. Hinze, Gh. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing - 11th International Conference, CMC 2010, Jena, Germany, August 24-27, 2010, Revised Selected Papers*, volume 6501 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2011.

Appendix A

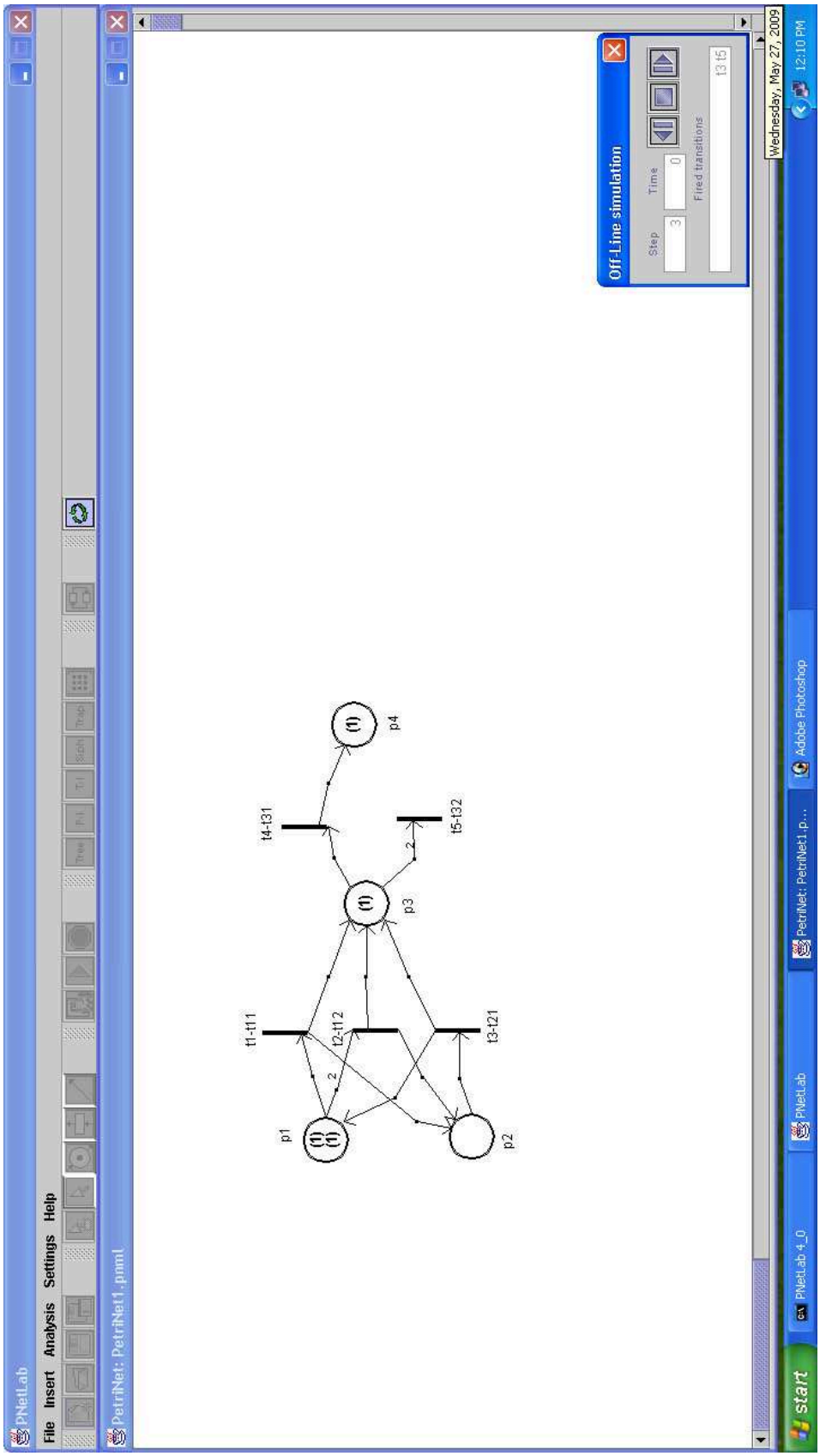
Step-by-Step Simulation of Petri Net $\mathcal{NL}_{\Pi_2}$ in PNetLab



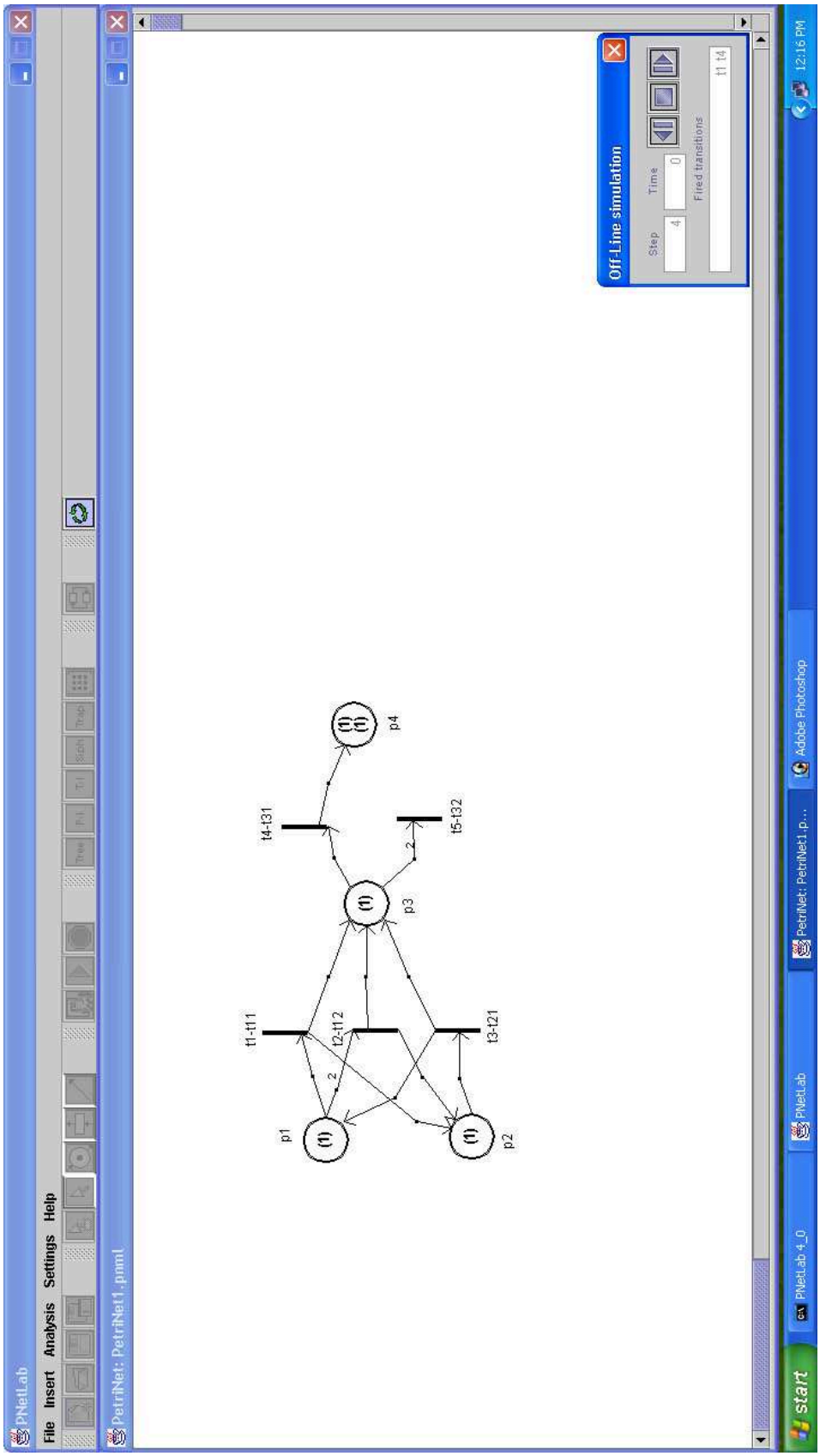
(a) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 1)



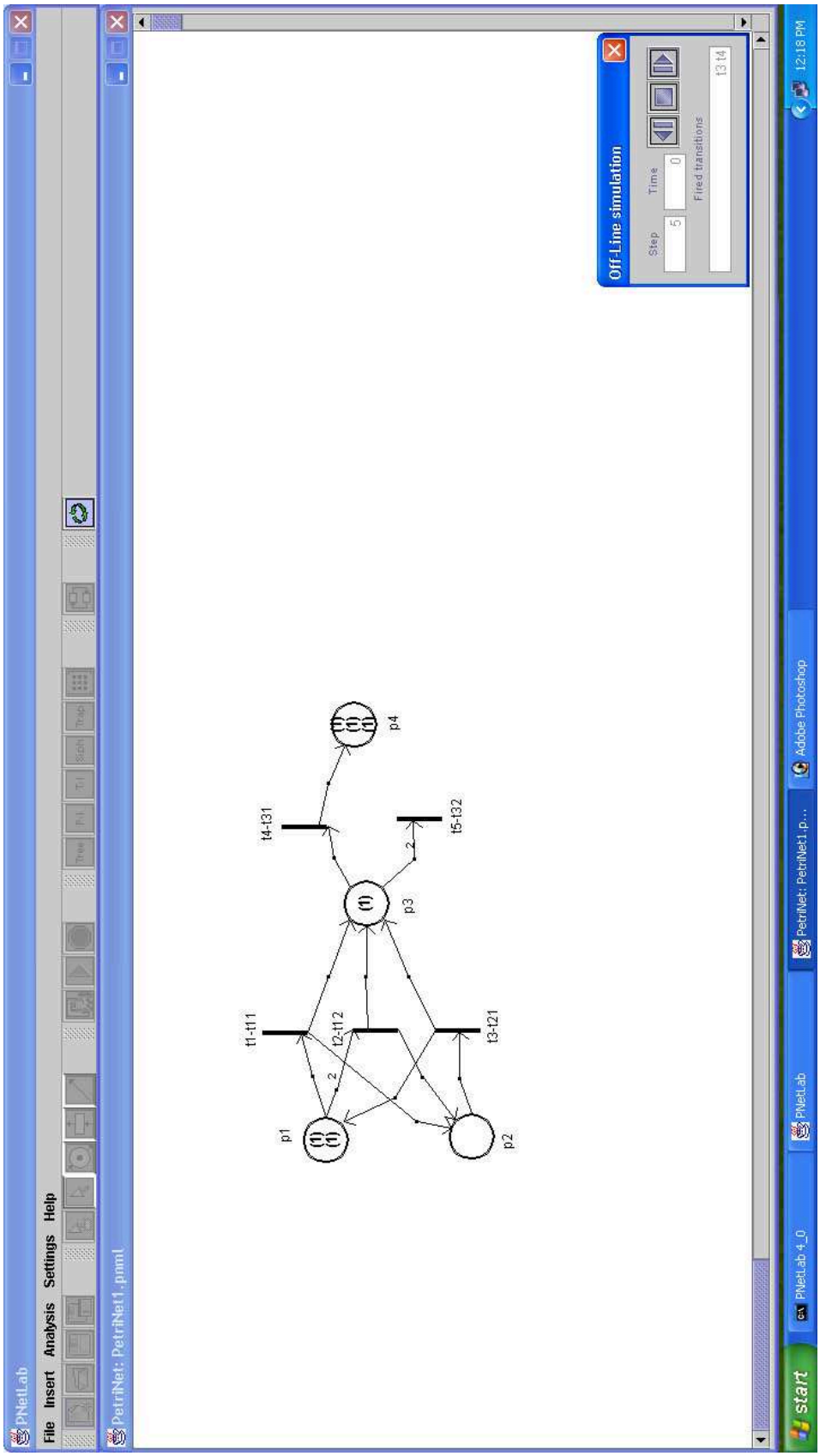
(b) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 2)



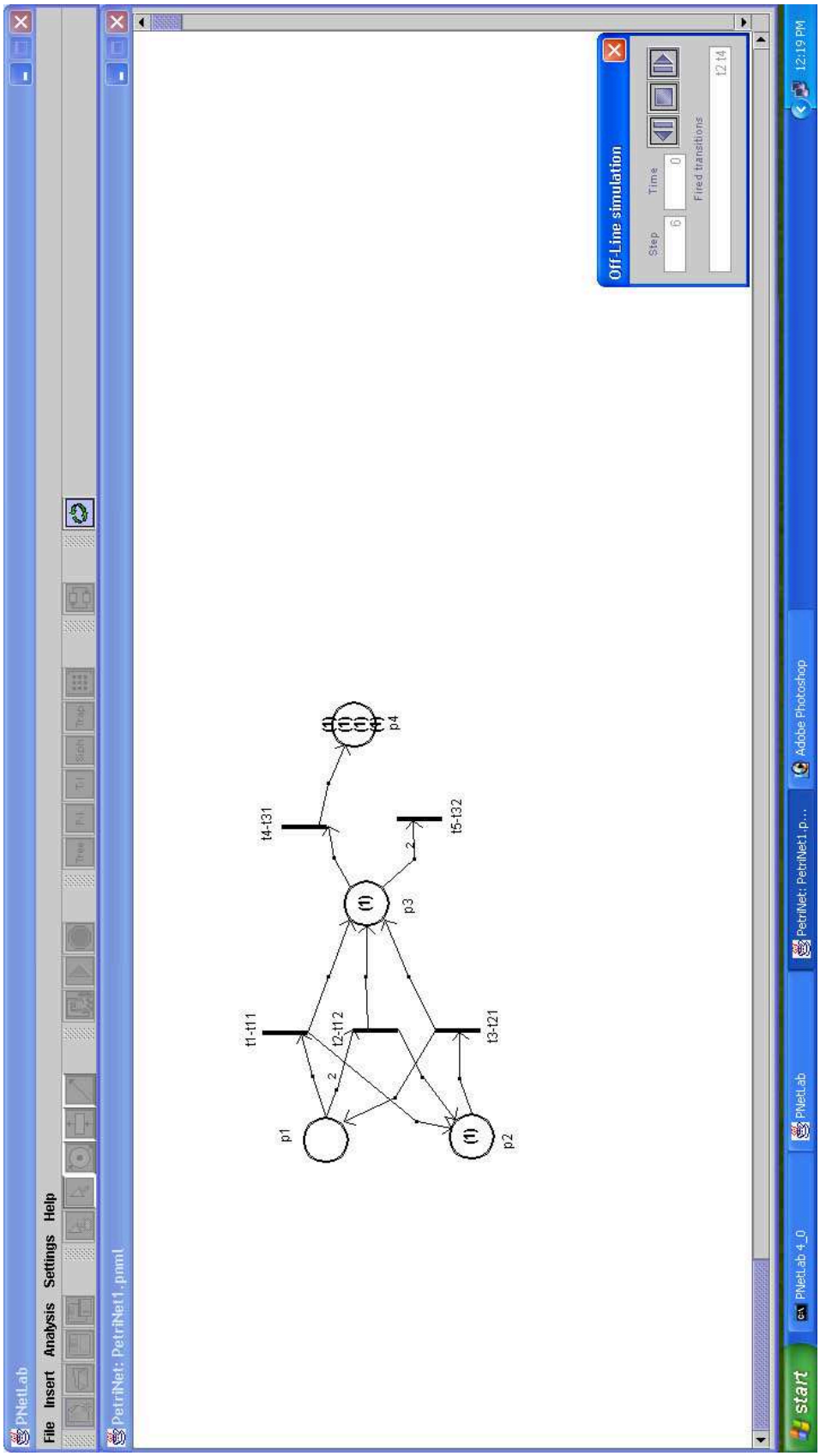
(c) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 3)



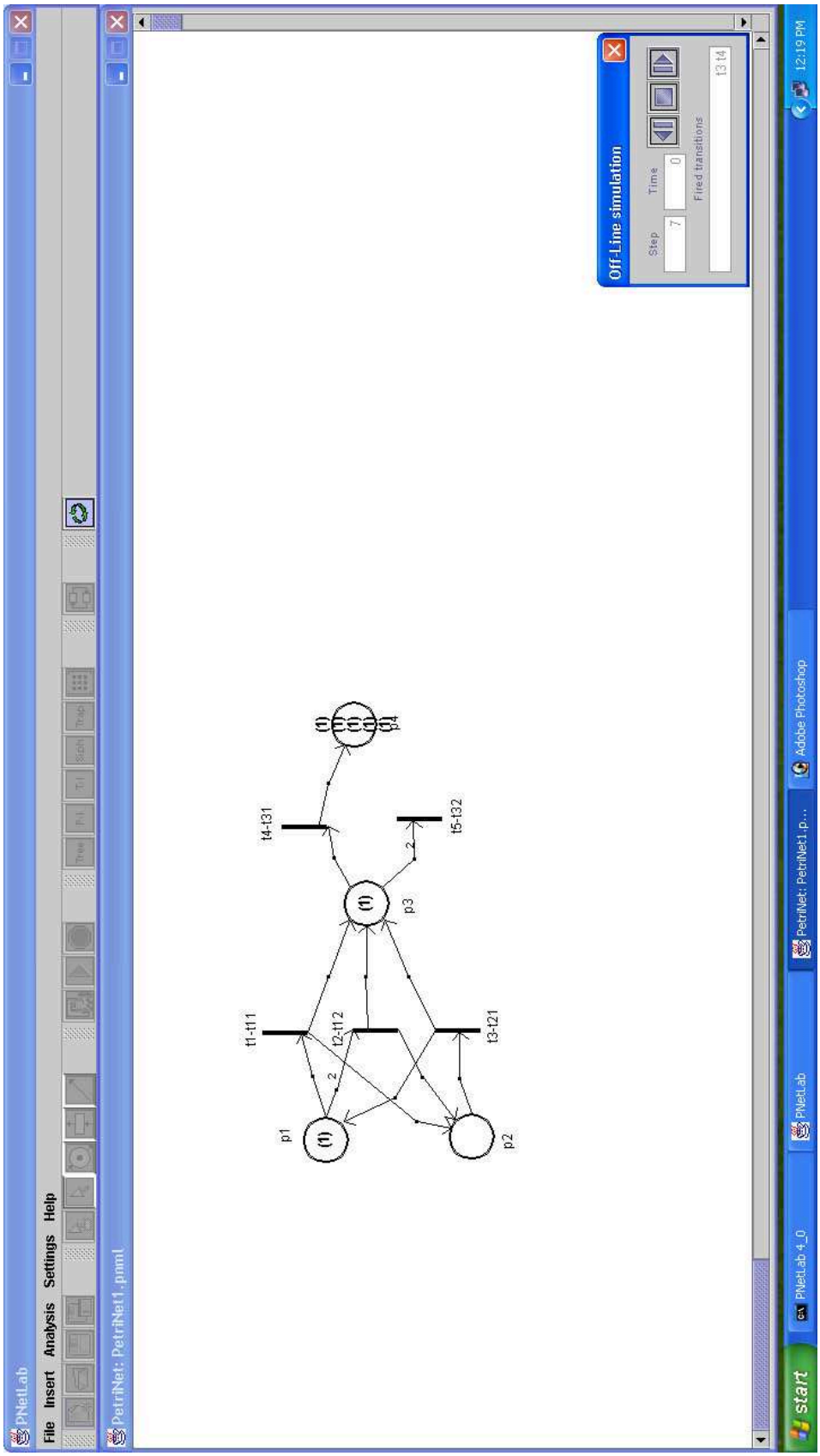
(d) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 4)



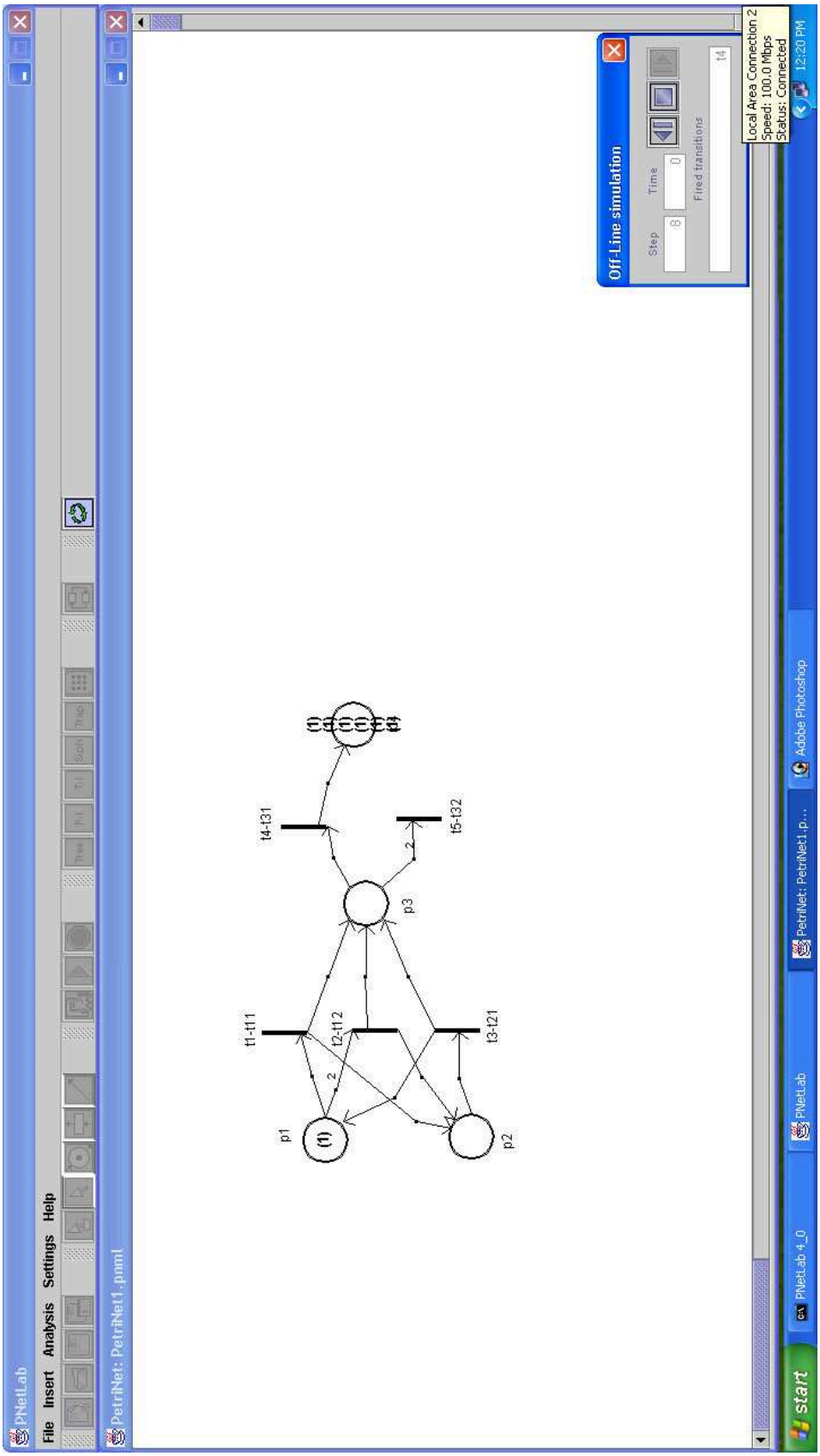
(e) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 5)



(f) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 6)



(g) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 7)

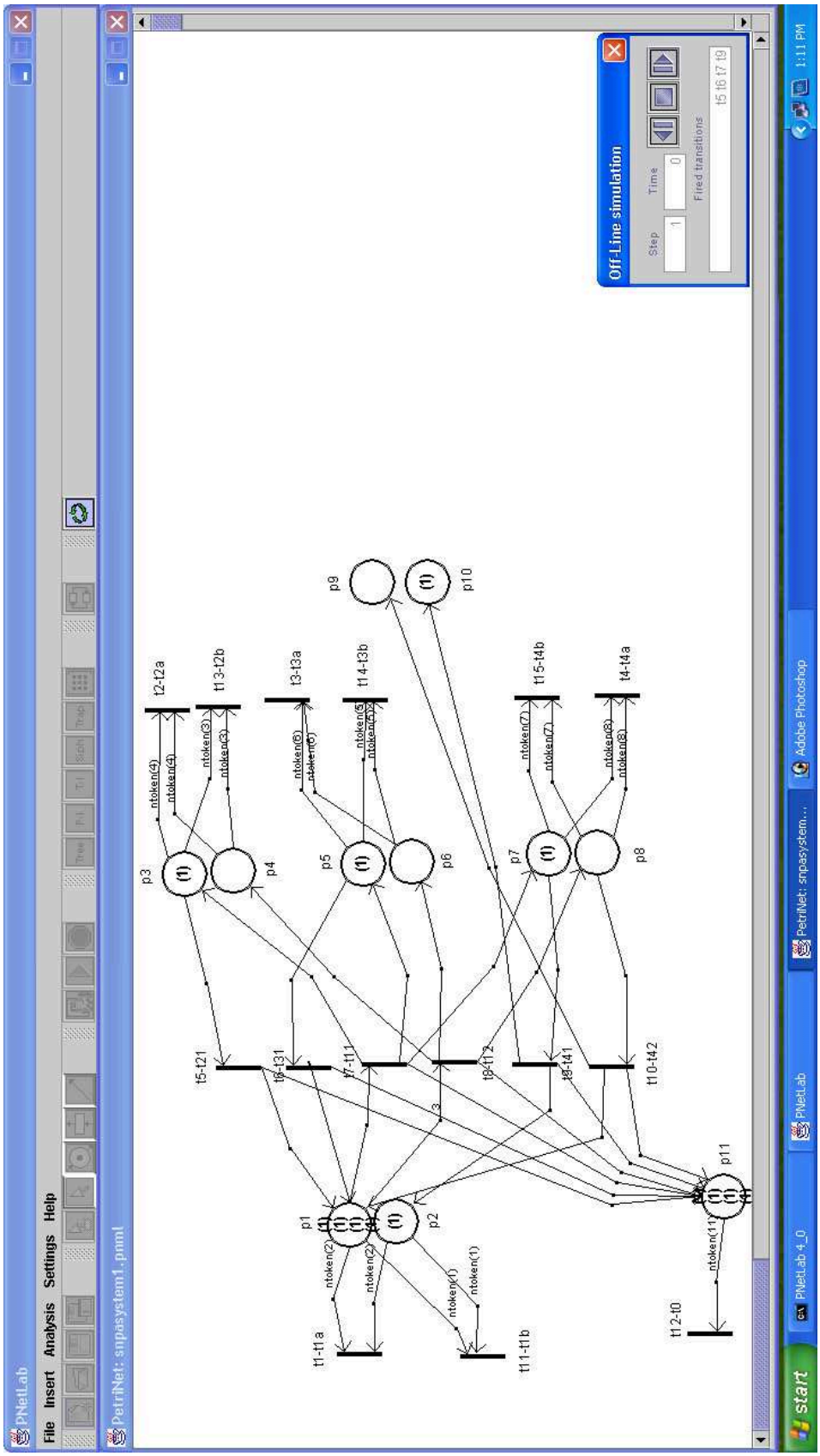


(h) Simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab (Step 8)

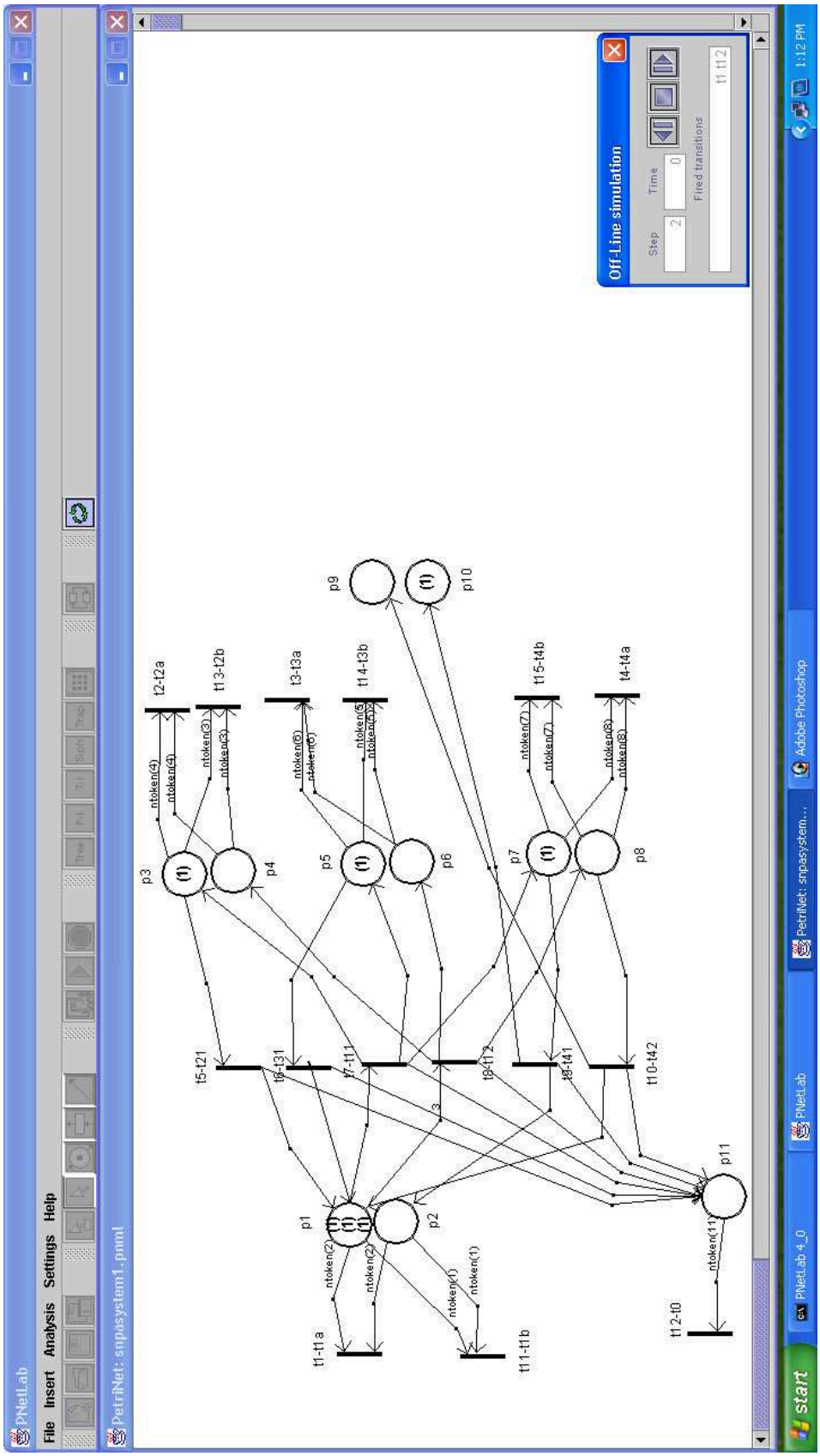
Figure A.1: Step-by-step simulation of $\mathcal{NL}_{\Pi_2}$ in PNetLab

Appendix B

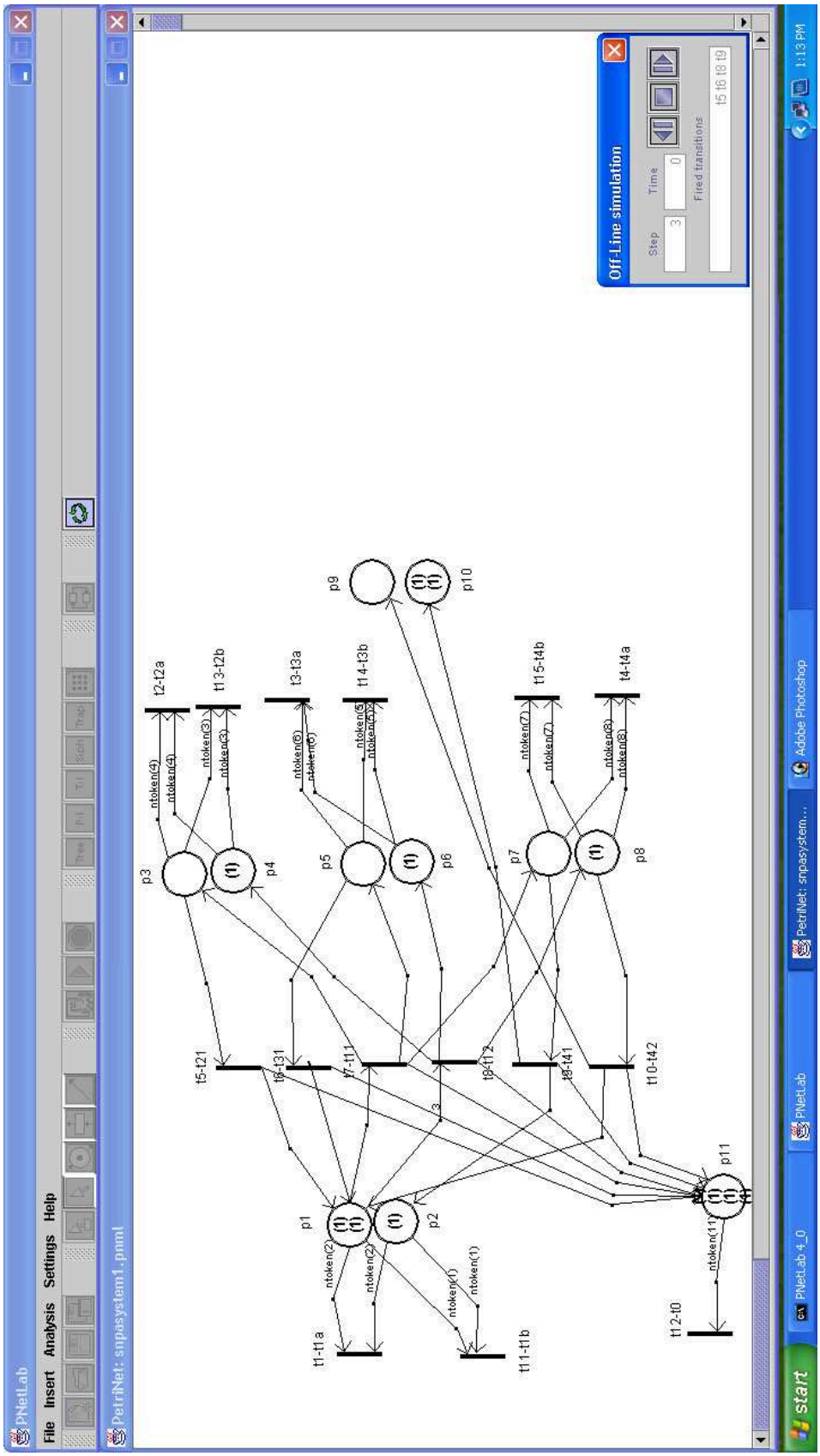
Step-by-Step Simulation of Petri Net $\mathcal{NL}_{\prod_3}$ in PNetLab



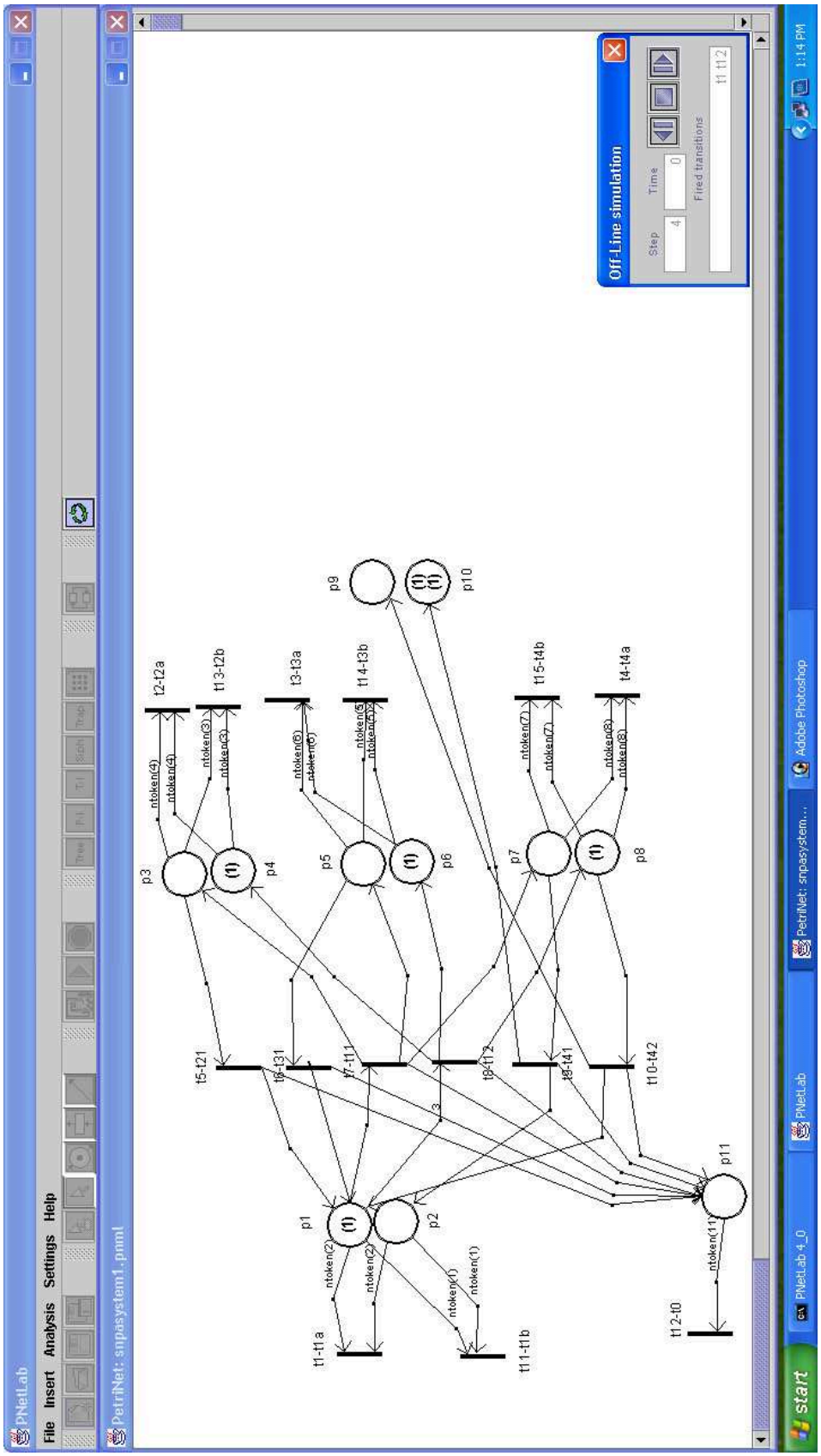
(a) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 1)



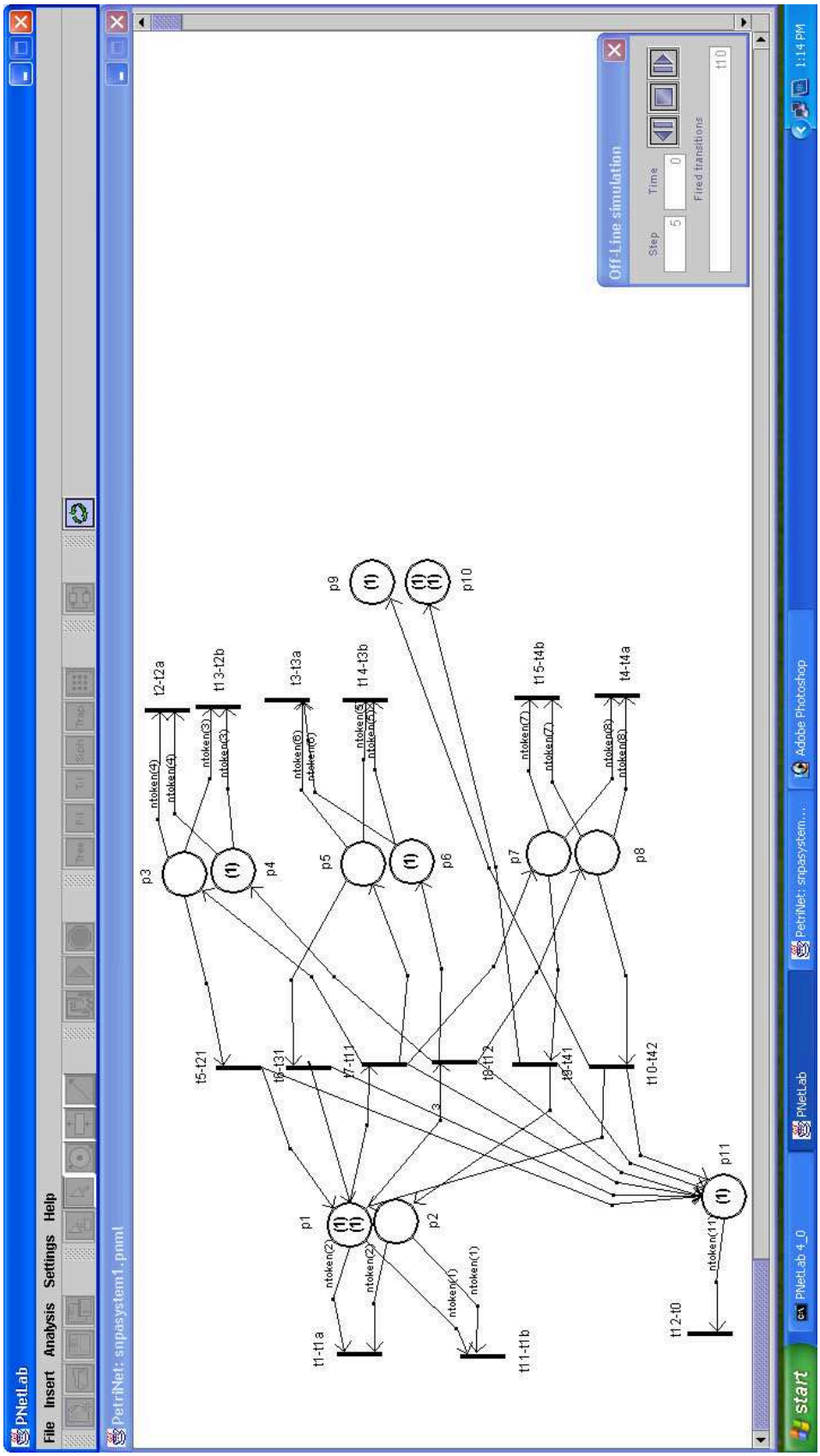
(b) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 2)



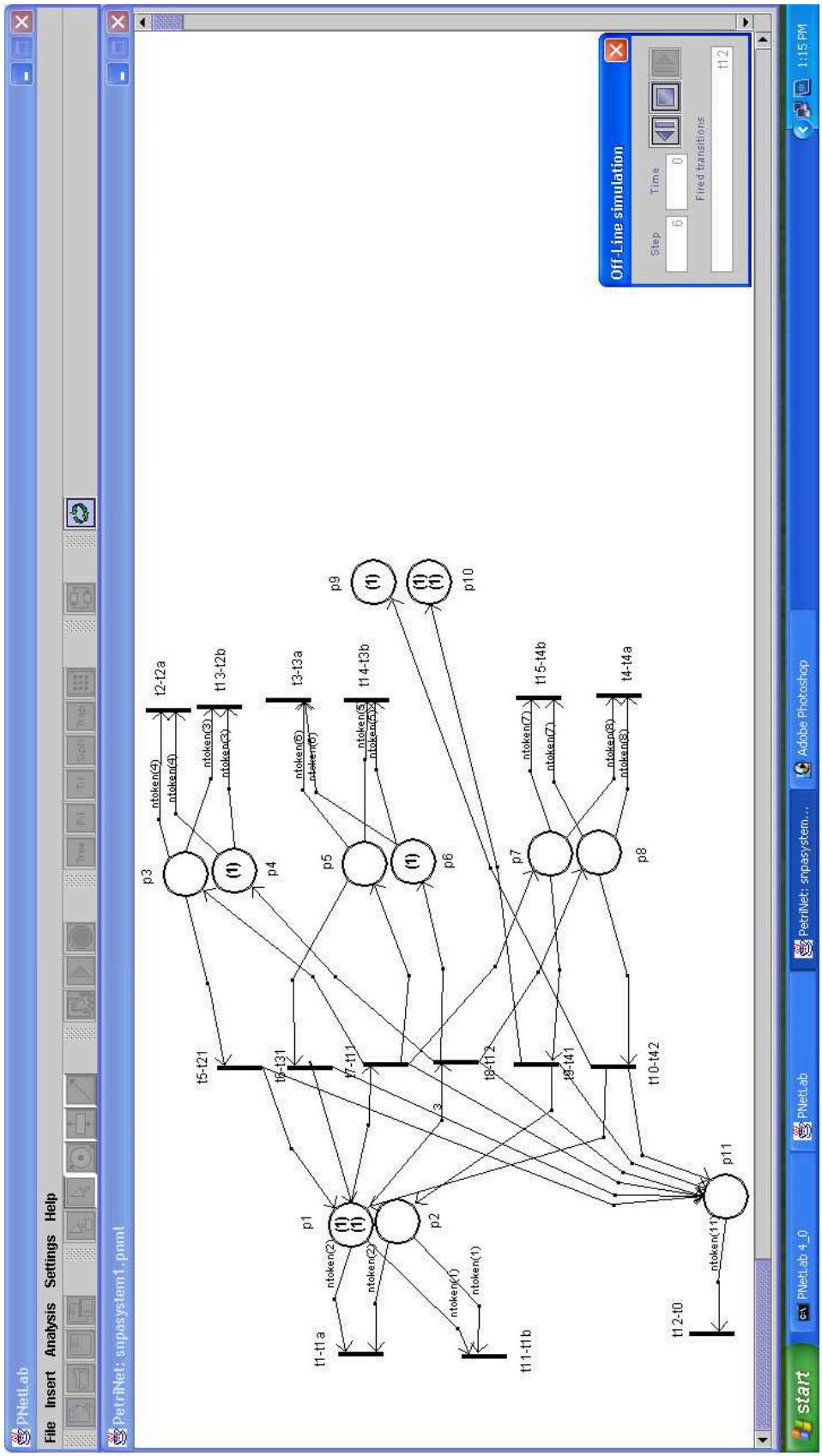
(c) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 3)



(d) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 4)



(e) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 5)



(f) Simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab (Step 6)

Figure B.1: Step-by-step simulation of $\mathcal{NL}_{\Pi_3}$ in PNetLab

List of Publications Based on the Thesis

Journals

1. V. P. Metta, K. Krithivasan, and D. Garg: Computability of Spiking Neural P Systems with Anti-spikes, *International Journal of New Mathematics and Natural Computation*, 8(3):283-295, 2012.

2. V. P. Metta, K. Krithivasan, and D. Garg: Spiking Neural P Systems with Anti-Spikes as Transducers, *Romanian Journal of Information Science and Technology*, 14(1):20-30, 2011.

3. K. Krithivasan, V. P. Metta, and D. Garg: On String Languages Generated by Spiking Neural P Systems with Anti-spikes, *International Journal of Foundations of Computer Science*, 22(1):15-27, 2011.

4. V. P. Metta, K. Krithivasan, and D. Garg: Modelling and Analysis of Spiking Neural P Systems with Anti-spikes using PNetLab, *Nano Communication Networks*, 2(2-3):141-149, 2011.

5. V. P. Metta, K. Krithivasan, and D. Garg: Protocol Modelling in Spiking Neural P Systems and Petri Nets, *International Journal of Computer Applications*, 1(24), 2010, http://www.ijcaonline.org /archives /number24/556-730.

Conferences

1. V. P. Metta, K. Krithivasan, and D. Garg: Simulation of Spiking Neural P Systems using PNetLab, *In M. Gheorghe, Gh. Păun, and S.Verlan, editors, Proceedings of the Twelfth International Conference on Membrane Computing (CMC-12), Fontainebleau, France*, pages 381-394, 23-26 August, 2011.

2. V. P. Metta, K. Krithivasan, and D. Garg: Representation of Spiking Neural P Systems with Anti-spikes through Petri nets, *In J. Suzuki, and T. Nakano, editors, Proceedings of the Fifth International ICST Conference on Bio-Inspired Models of Network, Information, and Computing System: BIONETICS 2010*, LNICST 87, pages 671-678, Springer, 2012.

3. V. P. Metta, K. Krithivasan, and D. Garg: Some Characteristics of Spiking Neural P Systems with Anti-spikes, *In M. Gheorghe, T. Hinze, and Gh. Păun, editors, Proceedings of the Eleventh International Conference on Membrane Computing (CMC-11), Jena, Germany*, pages 291-304, 24-27 August, 2010.

4. V. P. Metta, K. Krithivasan, and D. Garg: Computability of Spiking Neural P systems with Anti-Spikes, Informal Presentation, *Computability in Europe, CiE-2010*, University of Azores, Portugal, June 30- July 4, 2010.