Computations in Spiking Neural P Systems: Simulations and Structural Plasticity

Dissertation by

Francis George Carreon Cabarle
Master of Science in Computer Science

Submitted to the National Graduate School of Engineering
College of Engineering
University of the Philippines

In Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
in Computer Science

National Graduate School of Engineering
College of Engineering
University of the Philippines Diliman
Quezon City

June 2015

This dissertation, entitled COMPUTATIONS IN SPIKING NEURAL P SYSTEMS: SIMULATIONS AND STRUCTURAL PLASTICITY, prepared and submitted by FRANCIS GEORGE CARREON CABARLE, in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE is hereby accepted.

HENRY N. ADORNA, PhD
Dissertation Adviser

Prospero C. Naval Jr., PhD
Dissertation Panel Chairman

Jaime D.L. Caro, PhD
Dissertation Panel Member

Pablo R. Manalastas Jr., PhD
Dissertation Panel Member

Jan Michael C. Yap, PhD
Dissertation Panel Member

Mario J. Pérez-Jiménez, PhD
Dissertation Panel Member

Accepted as partial fulfillment of the requirements for the degree DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE.

AURA C. MATIAS, Ph.D.
Dean

UNIVERSITY OF THE PHILIPPINES

Doctor of Philosophy in Computer Science

Francis George Carreon Cabarle

Computations in Spiking Neural P Systems: Simulations and Structural Plasticity

Dissertation Adviser:
Henry N. Adorna, PhD
Department of Computer Science
University of the Philippines Diliman

Date of Submission:
June 2015

Permission is given for the following people to have access to this dissertation:

Available to the general public	Yes
Available only after consultation with author/dissertation adviser	No
Available only to those bound by confidentiality agreement	No

Signature of student

Signature of adviser

Acknowledgements

I subscribe to at least two (certainly not mutually exclusive) notions about life: that we are travellers (and sometimes students, teachers, etc.) and we each have our own travels in a journey called life, or that each one of us follows our own computations in a grand computation called life. Furthering the metaphors brought about by these notions, I am fortunate to have met many travellers during my computing journey, and many of them are also students, teachers, and *meaningful computations* in my opinion. I extend my deepest and most sincere gratitude to the following travellers who have (mostly) directly supported me technically or otherwise, in the completion of this present work.

I am primarily and largely indebted to the following: My advisor Henry N. Adorna, a wise traveller, student, and teacher (among other things) of research and life. His support, inspiration, and encouragements have allowed me to go further with my research and career. The Engineering Research and Development for Technology (ERDT) project of the DOST Science Education Institute of the Philippines has provided me with grants and opportunities used in the completion of this work and my degree. My singular travelling companion Sandy Mae, for her love, support, and for being a source of oracle-like inspirations. At the very least, I was wise enough to have chosen a wise woman (in more ways than one). My family, also for the love, support, and inspiration they provide me.

This work as well as my research journey would be incomplete without the generosity of the following colleagues, friends, collaborators: members of the Algorithms and Complexity lab, specifically Rich, Jasmine, Kelvin, Jhoirene, Nestine, Jan, Neil. El grupo de investigación en computación natural, dirigido por Mario de Jesús Pérez-Jiménez de la Universidad de Sevilla, y otros miembros, especificamente Miguel Ángel, Luis Felipe, Luis, Agustín, Álvaro, Carmen. Los investigadores de China: Tao Song, Tao Wang, Bosheng Song.

Some staff from UP Diliman: ma'am Joy and Dae (ERDT), ma'am Lynne (engineering graduate office), ate Mila and Grace (Computer Science).

I would also like to thank the inspirations, comments, and support from the following: Gheorghe Páun, Rudi Freund, Xiangxiang Zeng, and Sergiu Ivanov.

I also thank those I did not personally meet or whose computations have already halted, but who have certainly left their mark in life, so that others, myself included, could build upon their work. *If a dwarf could see further than giants, it is because the former is perched on the shoulders of the latter.*

Abstract

CARREON CABARLE, FRANCIS GEORGE. COMPUTATIONS IN SPIKING NEURAL P SYSTEMS: SIMULATIONS AND STRUCTURAL PLASTICITY. (Under the direction of HENRY N. ADORNA, PhD) *Spiking neural P systems* (in short, *SNP systems*) are parallel, distributed, synchronous, and nondeterministic models of computation. SNP systems are membrane models within the *membrane computing* area (a branch of the larger *natural computing* area) initiated by Gheorghe Păun in 1998. SNP systems are *inspired* by the structure and function of *spiking neurons*, which compute using indistinct signals called *spikes*. Neurons are spike processors which are placed on the nodes of a directed graph, where the directed edges between neurons are called *synapses*.

In this work, the neuroscience feature of *structural plasticity* is introduced into the SNP systems framework. The computing power of *SNP systems with structural plasticity* (in short, *SNPSP systems*) are investigated by simulating other models of computation, e.g. linear grammars, register machines. SNPSP systems are SNP systems that can create or delete synapses: they have a dynamical structure applied only to the synapses. We prove the computational (non)universality of SNPSP systems in several *semantics*, i.e. depending on how the system applies rules. Two of these semantics include the usual parallel and synchronous semantic, and a modification of this semantic called spike saving mode. The remaining semantics include two restrictions: removing either synchronization (i.e. *asynchronous* operation) or parallelism (i.e. *sequential* operation). We prove that such restricted SNPSP systems can still achieve universality. All universality results in this work apply to SNPSP systems as number generators or acceptors. In SNPSP systems, the *plasticity rules* used provide a new source of nondeterminism when selecting which synapses to create or delete. Plasticity rules also provide a "programming capacity" that could, at certain cases, replace two common rule types in SNP systems: *forgetting rules* (rules that remove but do not produce spikes) and *rules with delays* (a time delay takes effect prior to spike production). This nondeterminism source and capacity also allow SNPSP systems to have *normal forms*, which are a set of simplifying restrictions on system parameters, e.g. number or types of rules inside a neuron. We provide *uniform* modules of sequential SNPSP systems, i.e. the system construction is independent from the simulated register machine, providing some hint to an open problem on uniform modules of sequential SNP systems. The asynchronous SNPSP systems in this work provide support to the conjecture that asynchronous SNP systems with *standard rules* (each step, each neuron produces at most one spike) are not universal. Lastly, *semi-uniform* (i.e. system construction is based on the problem instance) and uniform solutions, both nondeterministic and under a normal form, to the **NP**-complete problem `Subset Sum` are provided. The uniform solution reduces the number of neurons by a linear amount (with respect to the input size) as compared to a previous uniform solution using SNP systems.

Keywords: *Natural computing, Membrane computing, Spiking neural P systems, Models of computation, Turing universality, Asynchronous systems, Sequential systems*

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

© Francis George Carreon Cabarle
Quezon city, Philippines, June 2015

Table of Contents

Acknowledgements **iv**

Abstract **v**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**

I Preliminaries **7**

2 Technical prerequisites **8**
2.1 Formal language theory 8
2.2 Machine models 10

3 Membrane computing **12**
3.1 Introduction 12
3.2 P system elements 14
3.3 Computing power and efficiency 15
3.4 Applications 16

4 Spiking neural P systems (SNP systems) **18**
4.1 Introduction 18
4.2 SNP systems syntax and semantics 19
4.3 Computing power 22
4.4 Other bio-inspired variants 23
4.5 Computing efficiency and applications 24

II Body of Work **26**

5 Notes on delays in SNP systems **27**
5.1 Introduction 27
5.2 Results 29

6 Notes on SNP systems and finite automata **34**
6.1 Introduction 34
6.2 DFA and DFST simulations 35
6.3 k-DFAO simulation and generating automatic sequences 39

7 SNP systems with structural plasticity (SNPSP systems) 42
7.1 Introduction . . . 42
7.2 Spiking neural P systems with structural plasticity . . . 44
7.3 An example of an SNPSP system . . . 46
7.4 Universality of SNPSP Systems . . . 47
7.5 Spike saving mode universality and deadlock . . . 53

8 Sequential SNPSP systems induced by max/min spike number 57
8.1 Introduction . . . 57
8.2 Sequential SNPSP systems based on spike number . . . 58
8.3 Main results . . . 60
8.3.1 Sequential SNPSP systems based on max spike number . . . 60
8.3.2 Sequential SNPSP systems based on min spike number . . . 64

9 Asynchronous SNPSP systems 68
9.1 Introduction . . . 68
9.2 Asynchronous SNPSP systems . . . 68
9.3 Main results . . . 70

10 Solving `Subset Sum` using SNPSP systems 75
10.1 Introduction . . . 75
10.2 A semi-uniform solution to `Subset Sum` . . . 77
10.3 A uniform solution to `Subset Sum` . . . 79

III Final Remarks 81

11 Conclusions 82

12 Further work 86

List of References 90

A Open and online resources for illustrations 97

List of Figures

2.1 2-DFAO generating the Thue-Morse sequence. 10

3.1 Four bio-inspired domains of natural computing. 13
3.2 Membrane structure of a cell-like P system with 8 membranes (including the skin), 5 of which are elementary. 14

4.1 A simple example of an SNP system generating the set $\mathbb{N}^+ - \{1\}$. 21

5.1 SNP system with delay Π_0. 28
5.2 Routing modules (from left to right): sequential, split, and join. 28
5.3 Sequential routing: Π_1 (top) with delay d, and $\overline{\Pi}_1$ (bottom) route simulating Π_1. 30
5.4 Sequential routing with multiple delays: Π_2 (top) with delays d_1 and d_2, and $\overline{\Pi}_2$ (bottom) route simulating Π_2. 30
5.5 Iteration routing: Π_3 (top) has delay d route simulated by $\overline{\Pi}_3$ (bottom). 32
5.6 Join routing : Π_4 (top) has delay d route simulated by $\overline{\Pi}_4$ (bottom). 33

6.1 Structure of SNP modules from [43] simulating DFAs and DFSTs. 36
6.2 DFA with incorrect simulation by the SNP module in Figure 6.3. 38
6.3 SNP module with incorrect simulation of the DFA in Figure 6.2. 38
6.4 SNP module simulating the 2-DFAO in Figure 2.1. 41

7.1 An SNPSP system Π_{ex}. 46
7.2 Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$. 49
7.3 Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$. 50
7.4 Module FIN. 51
7.5 Module INPUT. 52
7.6 Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j)$. 52
7.7 Module FIN$'$. 54
7.8 Module SUB$'$ simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$. 54
7.9 The SNPSP system $\Pi_{(+)}$ with a deadlock from the proof of lemma 7.5.1. 55
7.10 The SNPSP system Π' for the proof in Theorem 7.5.2. 56

8.1 Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.1. 61
8.2 Module SUB simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$ in the proof of Theorem 8.3.1. 61
8.3 Module FIN in the proof of Theorem 8.3.1. 63
8.4 Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.2. 64
8.5 Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.6. 66
8.6 Module SUB simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$ in the proof of Theorem 8.3.6. 66
8.7 Module FIN in the proof of Theorem 8.3.6. 66

9.1 An SNPSP system Π_{ej}. 69

9.2 Module ADD simulating $l_i : (\texttt{ADD}(1) : l_j, l_k)$ in the proof of Lemma 9.3.2. 71
9.3 Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$ in the proof of Theorem 9.3.2. 73
9.4 Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$ in the proof of Theorem 9.3.2. 73
9.5 Module FIN in the proof of Theorem 9.3.2. 74

10.1 The semi-uniform SNPSP system $\Pi_{\texttt{ss}}$ solving `Subset Sum`. 78
10.2 The uniform SNPSP system $\Pi_{\texttt{us}}$ solving `Subset Sum`. 80

List of Tables

5.1 Sample computations of Π_1 and $\overline{\Pi}_1$, $d = 3$. 30
5.2 Sample computations of Π_2 and $\overline{\Pi}_2$, $d_1 = 2$, $d_2 = 3$. 31
5.3 Sample computations of Π_4 and $\overline{\Pi}_4$, $d = 3$. 32

7.1 Computation of Π_{ex} computing $\{1\}$. 47
7.2 Computation of Π_{ex} computing $\{4\}$. 47

11.1 Summary of main (non)universality results concerning SNPSP systems in this work. 84

Chapter 1

Introduction

"If at first the idea is not absurd, then there is no hope for it."
Albert Einstein, quoted in Des MacHale, *Wisdom* (London, 2002).

One of the important lessons throughout human history is that the natural world has been a constant source of inspiration, awe, and even horror, especially if we do not have a fair understanding of a phenomena. The Earth for example, has been performing geological, chemical, biological (among others) spectacles for billions of years now. With respect to this scale of billions of years, the human species has existed for only a tiny fraction: scientists Carl Sagan and Neil deGrasse Tyson emphasized that in a "cosmic calendar" where the 1^{st} of January is when the Big Bang occurred, then the first signs of life (prokaryotic cells) only emerged on September, while the earliest known primates only emerged on the 30^{th} of December. In this calendar, the first known human writings (which marked the end of prehistory and the start of history) only occurred on the 31^{st} of December at 23 hours, 49 minutes, and 47 seconds. Clearly, there is still much more to be inspired from.

Many ancient civilizations respected the power of nature so that they built their societies to emphasize such power. One such civilization that is well known is that of the ancient Greeks, from whom many of the modern ideas of philosophy and science originated. The ancient Greeks of course appreciated nature, clearly shown by their pantheon of gods and goddesses. Their appreciation also manifested in their ideas, and in particular their science: observe, respect, and learn from nature. A good example perhaps of obtaining an abstraction from astronomical observation is an ancient analog computer called the Antikythera mechanism. This mechanism could, for example, predict eclipses and other astronomical positions which are useful for sea or land navigations. Such a mechanism was clearly a marvelous scientific achievement during that age.

From ancient civilizations, we fast forward to the modern era where the astonishing tool known as the computer has shaped the information revolution, following the industrial revolution. For many people, it is easy to dismiss the connection between inspiration from nature and the wondrous computer that has profoundly affected human life. It must be recalled however that the "giants"

and pioneers of the area of computation include scientists[1] who were inspired by nature: G.W. Leibniz for example is considered to be one of the first to attempt to formalize algorithms and computation;[2] In the work of A.M. Turing (see [92]) he observed (as Leibniz did before him) that human computers, e.g. bank clerks, are useful inspirations for modelling mechanical computation;[3] McCulloch and Pitts (see [56]) identified the brain as a fantastic repository of computing ideas, as did Turing (see [93]) and Kleene (see [50]); von Neumann, inspired by self-replication in biology (see [94]), conceived the idea of cellular automata together with Ulam.

The area known as *natural computing* continues this tradition of obtaining computing ideas from nature. Natural computing encapsulates the "classical" areas that include cellular automata, neural nets, and evolutionary computations (for a good overview article see [49]). Not only does natural computing obtain computing ideas from nature, some recent areas involve computations performed in nature itself, e.g. DNA computing (see [1]) and quantum computing (see [36]).

More recently introduced is *membrane computing*, by Gheorghe Păun, which started as a circulated article in 1998 before being published in 2000 (see [72]). Membrane computing (more details to follow in the succeeding chapters) started from the observation that the biological *cell* evolved, through billions of years, to become an astounding piece of "wet" machinery with many interesting ingredients: microscopic, power efficient, performing a vast number of operations in a parallel and distributed manner, (a)synchronously. The last three ingredients, together with nondeterminism (among others), are often used in membrane computing.

In this dissertation, the general aim is to continue this grand tradition of obtaining inspiration from nature for the advancement of science and the human condition. The contributions of this dissertation (provided shortly) fall under the sub-area of *spiking neural P systems* (in short, SNP systems) within membrane computing. In SNP systems, most of the computing inspirations and abstractions come from a particular type of cell that populate the brain, known as a neural cell or *neuron*. The modest contributions of this dissertation will hopefully provide ideas for computations inspired by the structure and functioning of neurons. The contributions in this work are in general theoretical in nature (along the lines of computability theory and hard problem solving) but always with the practical and biological interest and motivation to aid in the building of neural systems (whether in software, hardware, or wetware). In particular, in building theoretical or practical neural systems using the neuroscience phenomenon known as *structural plasticity*. This phenomenon allows neurons to create or delete *synapses* (the connections between neurons), thus changing the connectivity of neurons in the brain. The dissertation aims to provide a deeper understanding of computability in the framework of SNP systems with structural plasticity.

[1] Without whom a reasonable study of computation (at the very least its history) cannot really proceed.
[2] Following a principle of natural philosophy, *natura non facit saltum* i.e. "nature does not make a jump."
[3] The point of view being that humans are "natural objects" from nature.

Overview of the dissertation

The outline of the dissertation is provided next. The dissertation consists of several chapters distributed over three parts. Part I provides the necessary prerequisites for the remainder of this work. Part II provides the contributions of the dissertation (outlined shortly). Lastly, Part III concludes this work.

Part I: Chapter 2 provides the mathematical prerequisites used throughout this work: notions and notations from formal language theory, as well as machine models from computability and automata theory. Chapter 3 provides further details about membrane computing and relevant literature and results. Details on computability and complexity theory as applied to membrane computing are also given. Chapter 4 further expounds on the sub-area of SNP systems within membrane computing. Well known results on SNP systems, which are later referred to in this dissertation, are provided.

Part II: Chapter 5 provides constructions on how SNP systems with delays can be "route simulated" by SNP systems without delays. The objective is to allow both systems to be used as sub-modules for larger SNP systems that generate or accept numbers or strings. The route simulations are based on the idea of routing a spike in an SNP system module (with or without delay). This chapter is based on [10].

Chapter 6 provides some results regarding simulation of finite automata using SNP systems. Specifically, *SNP system modules* (in short, SNP modules) are used to simulate deterministic finite automata, deterministic finite transducers, and deterministic finite automata with output. SNP modules are also used to generate automatic sequences. This chapter is based on [14].

Chapter 7 begins the introduction of *SNP systems with structural plasticity*. Some motivations (computational and biological) are given, followed by a demonstrative example. Universality results for SNP systems with structural plasticity under a normal form (for two semantics) follow. This chapter is based on [13].

Chapter 8 and Chapter 9 continue the investigations started in the preceding chapter. Since SNP systems are parallel, nondeterministic, distributed, and synchronous devices, they are subjected to restrictions in order to further probe their limitations. The particular restrictions applied to SNP systems with structural plasticity are either (a) asynchronous operations, or (b) sequential operations. Universality results, even under normal forms, are provided. These chapters are based on [15] and [16].

Chapter 10 provides solutions to the **NP**-complete problem *Subset Sum* using SNPSP systems. The problem `Subset Sum` is solved using a semi-uniform solution and a uniform solution. Trade-offs for each solution type are given, as well as the idea of using synapse-level nondeterminism for variable generation and as an alternative to using delays or forgetting rules. This chapter is based in part on [13].

Part III: Chapter 11 provides conclusions obtained from the results in the preceding part of this

work. Lastly, Chapter 12 ends this dissertation by providing open problems and further research directions.

Contributions of the dissertation

In the following list and unless otherwise stated, the results for *computational universality* or simply universality (i.e. when a model can simulate any Turing machine) are implied to be for number generators (a number is generated from an initial configuration in a halting computation) or number acceptors (a number is introduced in the system and is accepted if the computation halts). Starting with Chapter 7, the conditions for (non)universality of SNP systems with structural plasticity (in short, SNPSP systems) are investigated. Two restrictions, which remove some of the "powerful" features of standard SNP and SNPSP systems (i.e. synchronous operation, and parallel operation) are applied to SNPSP systems. Again, the conditions when such restricted SNPSP systems become universal or not are provided.

- Constructions for *route simulating* SNP systems with delays using SNP systems without delays are provided. One consequence of these routing simulations is that more space (i.e. neurons) is required to simulate delays in SNP systems without delays.
- SNP modules are further investigated, with the following results:
  - The construction problem in simulating deterministic finite automata and transducers in [43] is amended. The amending construction also improves (i.e. reduces) the number of neurons in the SNP modules from three neurons (in [43]) to one neuron. This result is optimal in terms of the number of neurons in a module. SNP modules are also used to simulate k-DFA with output (in short, k-DFAO) using two neurons. Using these SNP modules that simulate k-DFAO, automatic sequences are then generated. Robustness properties of k-DFAO for generating automatic sequences (e.g. reading the input in reverse order) can then be transferred to the simulating SNP module.
- Introduction of SNPSP systems, which is a response to a challenge in [80] where "dynamism" in SNP systems is achieved only for synapses. Further results are as follows:
  - SNPSP systems are proven to be universal, even with a *normal form* (a simplifying set of requirements): SNPSP systems as number generators require nondeterminism, while determinism suffices for number acceptors. A (seemingly minor) change in the semantics of plasticity rule application in SNPSP systems, known as the spike saving mode, allows SNPSP systems in such a mode to maintain universality. In spike saving mode however, SNPSP systems can reach a *deadlock configuration* where the system is "stuck" (no further computations can proceed) and no output is produced. It is proved that reaching

such a configuration is undecidable with at least two neurons in the system.

- The restriction of sequential operation, induced by either the maximum or minimum spike number among the neurons in the system is imposed on SNPSP systems. Such restriction removes the parallel feature of SNPSP systems. We study four modes: max, max-pseudo, min, and min-pseudo sequentiality, with the following results:
  - We prove that as acceptors, deterministic SNPSP systems as acceptors are universal in all modes, while generators need a source of nondeterminism for max-pseudo and min-pseudo modes (provided in this work by synapse-level nondeterminism). With the use of synapse-level nondeterminism, we can construct a family of uniform modules for either max and max-pseudo, or min and min-psedo; constructing uniform families of modules is also a concern when introducing SNP systems variants. The sequential SNPSP systems considered here are also universal despite a normal form
- The restrction of asynchronous operation is imposed on SNPSP systems: we remove another powerful computing feature, the implicit global clock that synchronizes the functioning of neurons in the system. The results (for number generators only) are as follows:
  - (A1) Asynchronous bounded SNPSP systems, i.e. there is a given bound on the spikes that any neuron can accumulate, are proven to be not universal; (A2) Asynchronous SNPSP systems with weighted synapses, i.e. each synapse can have a weight of at least 1, and without bounding the accumulated spikes in a neuron are proven to be universal; Results (A1) and (A2) provide support to the conjecture of the still open problem of whether asynchronous SNP systems with standard rules are universal.[4]
- Semi-uniform and a uniform solutions to the problem `Subset Sum` using SNPSP systems are provided. We show how synapse-level nondeterminism can be used to reduce the needed neurons, especially for nondeterministic generation of variables. Both solutions run in constant time, while being under a normal form.

Selected publications of the author

- Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Asynchronous spiking neural P systems with structural plasticity. (to appear) 14th Unconventional Computation and Natural Computation, Auckland, New Zealand, and In: Calude, C., Dinneen, M. (eds.) LNCS vol. 9252 doi:10.1007/978-3-319-21819-9.
- Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Sequential spiking neural P systems with structural plasticity based on max/min spike number. (to appear) Neural

[4]The conjecture in [18] is that they are not.

Computing and Applications. doi:10.1007/s00521-015-1937-5 2013 IF: 1.763.

- Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J., Song, T. (2015) Spiking neural P systems with structural plasticity. (to appear, extended and improved version of [12]). Neural Computing and Applications. doi:10.1007/s00521-015-1857-4 2013 IF: 1.763.
- Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Notes on Spiking Neural P Systems and Finite Automata. (to appear) 13th Brainstorming Week on Membrane Computing, Sevilla, Spain
- Cabarle, F.G.C., Buño, K.C., Adorna, H.N. (2012) On The Delays in Spiking Neural P Systems. Philippine Computing Journal, vol. 7(2), pp. 12-17

Selected citations of the author

- Rosini, B., Dersanambika, K.S. (2014) Simulation of Boolean Circuits using Spiking Neural P Systems with Structural Plasticity. J. of Computer Trends and Technology. vol. 11(1), pp. 1-9
- Pan, L., Song, T. (2015) A Normal Form of Spiking Neural P Systems with Structural Plasticity. (forthcoming) J. of Swarm Intelligence

Lastly, we end this chapter by quoting an extract from The Handbook of Natural Computing in [88], which will be relevant throughout this work: "We are now witnessing exciting interaction between computer science and the natural sciences. While the natural sciences are rapidly absorbing notions, techniques and methodologies intrinsic to information processing, computer science is adapting and extending its traditional notion of computation, and computational techniques, to account for computation taking place in nature around us. Natural Computing is an important catalyst for this two-way interaction."

Part I

Preliminaries

Chapter 2

Technical prerequisites

In this chapter, we present mathematical and theoretical computer science prerequisites to be used in the remainder of this work. Familiarity with the basics of formal language, automata, and computability theory is assumed. Monographs, books, and resources on such topics are widely available in print and online, e.g. [37]. Notions and notations from these areas are mentioned here, however briefly.

2.1 Formal language theory

Standard set-theoretic mathematical notations are used in this work, e.g. $\emptyset$ is the empty set, an element a belonging to a set M is denoted as $a \in M$, inclusion of a set M in a set N is denoted as $M \subseteq N$, with strict inclusion written as $M \subset N$; The union, intersection, difference, and Cartesian product of sets M and N are denoted as $M \cup N$, $M \cap N$, $M - N$, and $M \times N$, respectively. The set of all natural (counting) numbers is denoted as $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, where $\mathbb{N}^+ = \mathbb{N} - \{0\}$ denotes the set of all positive integers. A *multiset* over a given set V is a mapping $M : V \to \mathbb{N}$.

An *alphabet* V is a finite non-empty set of abstract symbols. The set of all finite strings over an alphabet V forms a monoid with respect to the concatenation operation and the identity element (also the empty string) λ. This monoid is denoted as V^*, also known as a free monoid over V. The set of all non-empty strings over V (i.e. the free semigroup over V) is denoted as V^+, so $V^+ = V^* - \{\lambda\}$. V is called a *singleton* if $V = \{a\}$, and we simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. The *length* of a string $w \in V^*$ is denoted by $|w|$. If a is a symbol in V, the empty string λ is equal in meaning to a^0. If V has k symbols, then $[w]_k = n$ is the base-k representation of $n \in \mathbb{N}$. Each subset of V^* is a *language* over V, and a set or collection of languages is called a *family of languages*.

For a language $L \subseteq V^*$, the *length set* of L is defined as $length(L) = \{|x| \mid x \in L\}$. If FL is a family of languages, then we denote by NFL the family of length sets of languages in FL.

Regular languages can be defined by (among others) *regular expressions*. A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations

union, concatenation, and Kleene +, using parentheses whenever necessary to specify the order of operations. Specifically, (i) λ and each $a^k \in V$, for $k \in \mathbb{N}$, are regular expressions, (ii) if E_1 and E_2 are regular expressions over V then $E_1 \cup E_2$, E_1E_2, and E_1^+ are regular expressions over V, and (iii) nothing else is a regular expression over V. Associated with each expression E is a language $L(E)$ defined in the following way: (i) $L(\lambda) = \{\lambda\}$ and $L(a^k) = \{a^k\}$ for all $a^k \in V$ where $k \in \mathbb{N}$, (ii) $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$, $L(E_1E_2) = L(E_1)L(E_2)$, and $L(E_1^+) = L(E_1)^+$, for all regular expressions E_1, E_2 over V. Unnecessary parentheses are omitted when writing regular expressions, and $E^+ \cup \{\lambda\}$ is written as E^*. A language $L \subseteq V^*$ is *regular* if there is a regular expression E over V such that $L(E) = L$.

A *grammar* is a device with a finite description that *generates* a language. A *rewriting system* involves an alphabet V, a set of *axioms* in the form of a subset A of V^*, and a set P of *rewriting rules* in the form $r : u \to v$, for u, v being strings from V^*. Given a string $w = w_1uw_2$ we can rewrite u to v using rule r, and we obtain the string $z = w_1vw_2$. We write $w \Rightarrow_r z$, or simply $w \Rightarrow z$ (called a *direct derivation step*) if there is no confusion. The reflexive and transitive closure of this relation is denoted as $\Rightarrow^*$. 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 set of *rewriting* (also called *production*) rules of the form $u \to v$, with $u, v \in (N \cup T)^*$ and u contains at least one nonterminal symbol. N is the *nonterminal alphabet*, T is the *terminal alphabet*, S is the *axiom* or start symbol, and P is the set of productions. The derivation relation $\Rightarrow$ is defined as in any rewriting system: a string $w \in (N \cup T)^*$ such that $S \Rightarrow_G^* w$ is called a *sentential form*. The language generated by G is written as $L(G)$ and is defined as

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

Example 2.1.1. *Given the grammar* $G = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to ab\})$, *the generated language is* $L(G) = \{a^nb^n | n \geq 1\}$.

Each derivation in G involves $m \geq 0$ applications of $S \to aSb$, and this produces strings of the form a^mb^m. The derivation halts after the rule $S \to ab$ is used, producing the string $a^{m+1}b^{m+1}$. We have that any string generated by G is of the form a^nb^n for some $n \geq 1$, and conversely, each string of this form can be generated by G.

A grammar is *linear* if each rule $u \to v \in P$ has $u \in N$, and $v \in T^* \cup T^*NT^*$. A special type of linear grammar is a *right-linear* grammar (also known as a right-regular grammar), where each rule $u \to v \in P$ has $u \in N$, and $v \in T^* \cup T^*N$. Left-linear grammars also exist, in this case however all nonterminal symbols appear on the left end of the right hand side of the rule. The family of languages generated by linear and regular (i.e. left- and right-linear) grammars is denoted as LIN, and REG, respectively. From the *Chomsky hierarchy* we have $REG \subset LIN$. However, when considering length sets of languages we have $NREG = NLIN$, since the length set of a linear language can be obtained as the length set of a regular language. A finite union of linear sets is also called a *semilinear set*, so that in this work we use $SLIN$ instead of $NREG$. The grammar in

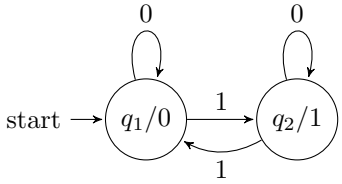

Figure 2.1: 2-DFAO generating the Thue-Morse sequence.

Example 2.1.1 is a linear grammar, but not a regular grammar (i.e. neither left- nor right-linear).

In this work we use a **convention**: when comparing the sets of numbers (i.e. length sets) of number generating or accepting devices in the latter sections, we ignore the number zero. This convention corresponds to the convention from language and automata theory of ignoring the empty string when comparing the languages generated (or accepted) by two devices.

2.2 Machine models

Next we define some well known computing models or devices used later in our proofs. These models include universal and non-universal models of computation. A *deterministic finite automaton* (in short, a DFA) D, is defined by the 5-tuple $D = (Q, \Sigma, q_1, \delta, F)$, where $Q = \{q_1, \ldots, q_n\}$ is a finite set of *states*, $\Sigma = \{b_1, \ldots, b_m\}$ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the *transition function*, $q_1 \in Q$ is the *initial state*, and $F \subseteq Q$ is a set of *final states*.

A *deterministic finite state transducer* (in short, a DFST) with accepting states T, is defined by the 6-tuple $T = (Q, \Sigma, \Delta, q_1, \delta', F)$, where Q, Σ, q_1, and F are the in a DFA, with Δ as the output alphabet, and the transition function now defined as $\delta' : Q \times \Sigma \to Q \times \Delta$.

A *deterministic finite automaton with output* (in short, a DFAO) M, is defined by the 6-tuple $M = (Q, \Sigma, \delta, q_1, \Delta, \tau)$, where Q, Σ, Δ, and q_1 are the same as in a DFST, with the transition function $\delta : Q \times \Sigma \to Q$, and $\tau : Q \to \Delta$ is the *output function*.

A given DFAO M defines a function from Σ^* to Δ, denoted as $f_M(w) = \tau(\delta(q_1, w))$ for $w \in \Sigma^*$. If $\Sigma = \{1, ..., k\}$, denoted as Σ_k, then M is a *k-DFAO*. A sequence, denoted as $\mathbf{a} = (a_n)_{n \geq 0}$, is *k-automatic* if there exists a k-DFAO, M, such that given $w \in \Sigma_k^*$, $a_n = f_M(w) = \tau(\delta(q_1, w))$, where $[w]_k = n$.

Example 2.2.1. *(Thue-Morse sequence) The Thue-Morse sequence* $\mathbf{t} = (t_n)_{n \geq 0}$ *counts the number of 1's (mod 2) in the base-2 representation of* n*. The 2-DFAO for* $\mathbf{t}$ *is given in Fig. 2.1.*

In order to generate $\mathbf{t}$, the 2-DFAO is in state q_1 with output 0, if the input bits seen so far sum to 0 (mod 2). In state q_2 with output 1, the 2-DFAO has so far seen input bits that sum to 1 (mod 2). For example, we have $t_0 = 0$, $t_1 = t_2 = 1$, and $t_3 = 0$.

A *register machine* (sometimes called a counter machine) is a construct $M = (m, I, l_0, l_h, R)$, where m is the number of *registers*, I is the finite set of *instruction labels*, l_0 is the *start label*, l_h is the *halt label*, and R is the finite set of *instructions*. Every label $l_i \in I$ uniquely labels only one instruction in R. Register machine instructions have the following forms, given the stored value n in register r:

- $l_i : (\texttt{ADD}(r), l_j, l_k)$, increase n by 1, then nondeterministically go to l_j or l_k;
- $l_i : (\texttt{SUB}(r), l_j, l_k)$, if $n \geq 1$, then subtract 1 from n and go to l_j, otherwise perform no operation on r and go to l_k;
- $l_h : \texttt{HALT}$, the halt instruction.

Given a register machine M, we say M computes or generates a number n as follows: M starts with all its registers empty. The register machine then applies its instructions starting with the instruction labeled l_0. Without loss of generality, we assume that l_0 labels an `ADD` instruction, and that the content of the output register is never decremented, only added to during computation, i.e. no `SUB` instruction is applied to it. If M reaches the halt instruction l_h, then the number n stored during this time in the first (also the output) register is said to be computed by M. We denote the set of all numbers computed by M as $N(M)$.

It is known (e.g. in [58] and [59]) that register machines compute all sets of numbers computed by a Turing machine, therefore characterizing NRE, the family of length sets of languages in RE (the set of recursively enumerable languages computed by Turing machines).

Register machines can also work in an accepting mode. A number n is stored in the first register of register machine M, with all other registers being empty. If the computation of M starting with this initial configuration halts, then n is said to be accepted or computed by M. In the accepting mode and even with M being deterministic, i.e. given an `ADD` instruction $l_i : (\texttt{ADD}(r), l_j, l_k)$ with $l_j = l_k$ written simply as $l_i : (\texttt{ADD}(r), l_j)$, register machine M can still characterize NRE.

A *strongly monotonic register machine* is one restricted variant of register machines: it has only one register which is also the output register. The register initially stores zero, and can only be incremented by 1 at each step. Once the machine halts, the value stored in the register is said to be computed. It is known that strongly monotonic register machines characterize $SLIN$, the family of length sets of regular languages. Since register machines are often easier to simulate than Turing machines, many P system proofs use this result (this is also done in this work).

Chapter 3

Membrane computing

This chapter presents a brief overview of membrane computing, including relevant information for the remainder of this work: Some background and motivations (e.g. biological, computational); Common elements and conventions; Known theoretical (e.g. along the lines of computability and complexity theory) and practical (e.g. simulations, modelling) results.

3.1 Introduction

As established previously in Chapter 1, there is still much to learn from nature for computing purposes. However and before we proceed, we always have the implicit assumption that *nature computes* or performs computations. By this assumption we simply mean that any transformational process occurring in nature, from input to output, can be considered as computation. Of course one of the main driving force for research in the area of natural computing is identifying the encoding to use, so that our human problems can be transferred to these transformational processes.[1] As Păun and Pérez-Jiménez pointed out in [74], "In some sense, the whole history of computer science is the history of continuous attempts to discover, study, and, if possible, implement computing ideas, models, paradigms from the way nature – the humans included – computes."

The physical limitations of computing *in silico* are some of the reasons why at present, instead of making processors operate faster, we put more processors or cores together side-by-side as parallel co-processors. Not only can we not keep making silicon-based transistors smaller and faster, their energy consumption and produced heat are also critical physical problems that are difficult, if not impossible, to overcome. Overcoming these limitations, as well as continuing to be inspired by natural processes, are some of the reasons for the vigorous activity in the area of natural computing. There exist various journals, book series, research groups, and annual conferences devoted to natural computing and its branches. For membrane computing, there exist comprehensive and introductory books in [73] and [22], more advanced ones in [23], [28], and a recent handbook in [81]. Membrane

[1] For further insights on the nature of computation and computation in nature (in particular in biology), its history and influence on computing, see [79] [87].

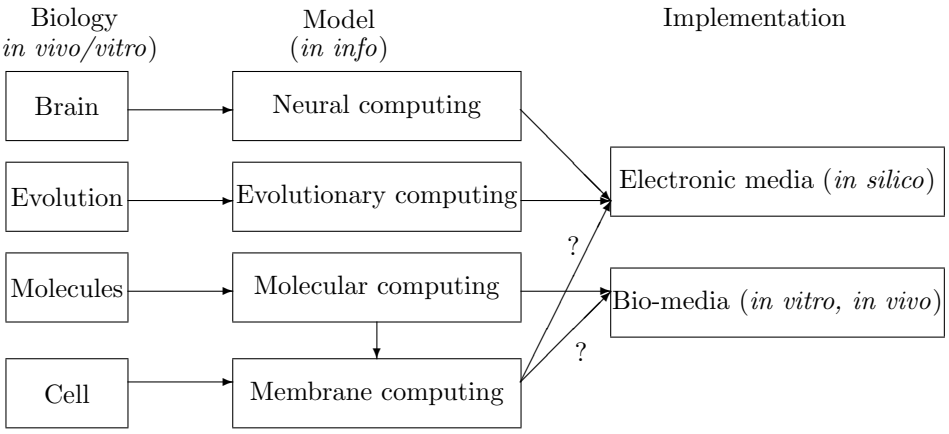

Figure 3.1: Four bio-inspired domains of natural computing.

computing is also included in a massive (four volumes) recent handbook on natural computing in [88]. The web page dedicated to collecting updated information on membrane computing, including a collection of PhD theses (61 at present writing time) is found in [105]. Also, the Institute for Scientific Information or ISI included membrane computing in their emerging research fronts in the field of computer science, as early as 2003 [104].

Figure 3.1 which is adapted from [73], shows four bio-inspired domains of natural computing. The edge connecting molecular computing towards membrane computing exists because membrane computing can be thought of as "zooming out" of the DNA molecules and considering the higher order cell structure. This connection is one reason why membrane computing is considered to be an extension of molecular computing. Computing models in membrane computing are called *P systems*, after their initiator Gheorghe Păun. The "wet lab" experiment of Adleman in [1] convinced the community that molecular computations specific to the DNA can be performed, thus birthing the area of DNA computing. P systems on the other hand, are yet to be fully implemented either *in silico*, *in vivo*, or *in vitro* (some reasons for this are given shortly).

Essentially, membrane computing is cell-inspired computation: the structure and functioning of the cell. The name membrane computing developed from the crucial role of membranes in cell structure and functioning: membranes compartmentalize processes and chemical reactions, as well as define regions outside the cell (the environment) and nested regions within the cell. In these "protected reactors", parallel, distributed, (a)synchronous transformations and communications of chemicals occur. The cell also does not seem to have the problems that plague *in silico* technologies mentioned earlier in this section. As early as 1990, the cell has been identified to be a rich source of ideas for parallel and distributed computations [4]. Membrane computing is thus a response to the challenge of obtaining formalisms and performing systematic investigations on computations inspired by the cell, largely untouched by molecular and DNA computing.

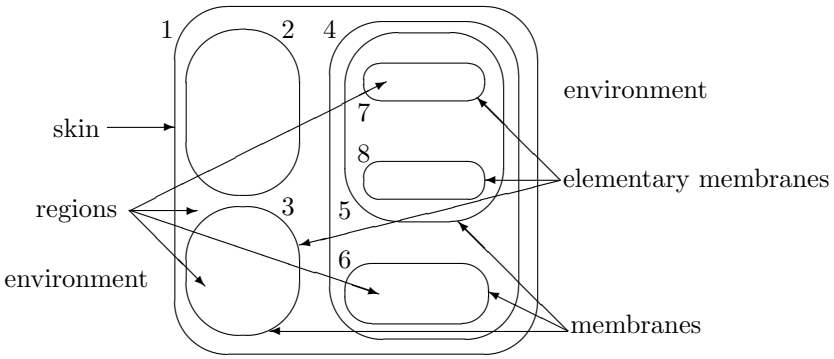

Figure 3.2: Membrane structure of a cell-like P system with 8 membranes (including the skin), 5 of which are elementary.

3.2 P system elements

Next, we mention common P system elements, i.e. syntax and semantic contents. Since the cell is a three-dimensional structure[2], abstracting it for formal use as a P system often involves creating two-dimenstional illustrations using ovals representing membranes as in Figure 3.2. This illustration in Figure 3.2 can of course be generalized to contain arbitrarily many membranes (the ovals) arranged in arbitrarily many levels. Chemicals inside or outside the cell are abstracted and simply called *objects*. Since the chemicals in the cell are assumed to be "swimming" in an aqueous solution, the most common data structure used in P systems is the *multiset* (a set with multiplicities associated with its elements). The basic idea is that similar to biological cells, each P system compartment allows chemical reactions to occur in a parallel and distributed manner.

The system can choose in a deterministic or nondeterministic way which reactions are allowed to occur. In this way, we can have computations defined as transitions from one configuration to another. A halting computation (where no reaction can take place) can provide a result: for example as the number of certain objects found in a certain membrane, or the existence of certain objects such as *yes* or *no* for decision problems.

The chemical reactions that transform chemicals (objects) are abstracted as multiset rewriting rules known as *evolution rules*. *Communication rules* are used to send (either directly or indirectly) or receive objects from one membrane to another. In Figure 3.2 for example, we have a tree-like structure where the skin membrane is the root and has label 1. The skin membrane is the outermost membrane delimiting the P system and its *environment*. Each membrane delimits a *region*, i.e. the area surrounded by the membrane. Hence there is a one to one correspondence between a membrane and a region. *Elementary membranes* are those which do not contain further membranes inside them. The environment usually contains an unbounded number of objects for the P system to use. Each region in a P system contains a set of rules of either type, which can be applied depending on the

[2]A list of online and open access illustrations of biological cells (e.g. structure, components) is in Appendix A.

availability of objects in that region and specific object requirement of each rule.

From this basic P system model, a menagerie of variants have been produced from various motivations, e.g. mathematically, biologically, etc. For example, the typical evolution rule $u \rightarrow v$ can be *cooperative* or *noncooperative*: the former is when at least two objects appear in multiset u (written as a string) while the latter is when u consists of only one object. Cooperative rules represent the process when chemical or object a reacts with object b to produce object c, while noncooperative rules represent singular object reactions. Communication rules can have target indications in v: a target *in* means an object is (non)deterministically transferred to an immediately inner membrane, if such a membrane exists, otherwise the rule cannot be applied; a target *out* means an object is sent out from the present membrane, thus becoming an object in the immediately surrounding region.

Besides these rule types, there exists various *execution strategies* for applying rules. One common strategy is *maximal parallelism*: aside from possibly choosing rules to apply in a nondeterministic way, all objects that can evolve or be communicated *must* do so. This strategy applies to every region of the system. Furthermore, in other P systems we can have rules that change the structure of the system. For example, we can have *membrane dissolution rules* which when applied, removes the membrane associated with the dissolution rule. All rules inside the dissolved membrane are also removed, but all objects are inherited by the previously surrounding region. We can also have *membrane division rules* which when applied, creates two new membranes with a replica of all the rules and objects (and sometimes even inner membranes) of the original membrane. Such membrane operations are biologically inspired from cell activities. Other P system variants also exist that take inspiration from other cell types and structures, e.g. cell tissues, colonies of bacteria. We do not go here into the details of such P systems and the reader is invited to seek for example [22] [23][28] [73] [74] [81] [105] and references therein.

3.3 Computing power and efficiency

Two of the important issues when defining computing models is *computating efficiency* and *power*. The former issue is concerned with questions related to how hard problems are efficiently solved, either by trading time for space or vice versa. The latter issue is concerned with what the model can compute, especially with respect to known models of computation.

One reason why faithful implementations of many (though not all) P systems on existing hardware or even bio-ware are difficult at present is that by default P systems are powerful computing devices[3] with features that are difficult to reasonably replicate with present technologies. A simple example, related to the computational efficiency of P systems with membrane division (also known as *P systems with active membranes*) is exponential creation of space in linear time: if the problem input size is n and we need to check 2^n candidate solutions, we start with a P system with active

[3]Which is why it is also not so surprising that most P system variants are universal devices, as will be further elaborated shortly.

membranes, having only the skin membrane and one elementary membrane inside; each step, the elementary membrane divides to produce two more elementary membranes; the succeeding elementary membranes would keep dividing, and so on; after n steps we will have 2^n membranes, which represent space and processors in P system parlance.

Conceptually at least, it is easy to see how P systems can solve **NP**-complete problems in polynomial or linear time. What is not so easy is to perform such exponential space creation (with or without other common P system ingredients, e.g. maximal parallelism) with our existing technologies. Furthermore, creating exponentially many membranes, and thus exponentially many objects, is necessary to be able to solve **NP**-complete problems according to the Theorem in [99] (unless $\mathbf{P} = \mathbf{NP}$). For further details on computational efficiency of P systems, efficient solutions to hard problems (e.g. `SAT`, Hamiltonian Path Problem), borderlines between efficiency and non-efficiency, and in relation to standard complexity measures, further details can be found in [22] [83] [99]. Much still remains to be considered and solved in these theoretical interests however, together with issues on descriptional complexity, i.e. the resources required to describe such systems.

With regards to the computing power of P systems, most variants are known to be universal. The reason for this is that the intuition, at least in language and automata theory, for a computing model to be as powerful as Turing machines requires the ability to (1) control computations in a precise way, and (2) erase or remove elements. Both abilities are available in abundant amounts in the cell: ability (1) assumes sufficient context-sensitivity, e.g. a reaction $u \to v$ where u involves several objects; ability (2) allows for an arbitrarily large workspace to be (re)utilized, e.g. by erasing objects as in $u \to \lambda$ or sending objects to a membrane designated for "garbage collection". For some P system variants, the cost of achieving universality comes at a minimal price. For example, simple *symbol-object* P systems, similar to the basic P system presented in Section 3.2, can achieve universality with at least one membrane, having cooperative rules, rules with targets, applied nondeterministically. A table with an early summary of universality results, as well as open problems, is available in an appendix in [73].

Common results as well as open problems with regards to the computability of P systems often come in two flavours: in generating or accepting (sets of) numbers or languages. As mentioned previously, generating a number can performed by counting the multiplicities of produced objects in a halting configuration. Accepting numbers involves introducing a number into the system, and then checking if the system halts. For languages, the system generates or accepts strings over the alphabet of objects.

3.4 Applications

As mentioned in Section 3.1 and aside from theoretically attractive properties of membrane computing, vigorous research activity is also owed to the attractive practical properties of P systems:

understandability (e.g. biologists can easily grasp evolution rules as chemical reactions), scalability and extensibility (e.g. further membranes, rules, or objects can be added without essentially changing the system functioning), modularity (e.g. entire P systems or membranes can be composed of individually independent systems or membranes acting as modules), programmability (e.g. using evolution or communication rules), and parallelism, just to name a few.

As previously mentioned, membrane computing was a response to a challenge of obtaining formalisms for computing use inspired by cells. Fortunately and because of the attractive properties of P systems, many researchers have taken up the challenge of applying P systems for practical applications. In [63] for example and extended in [64], the first optimization algorithm inspired by P systems was introduced. P systems have also been used as (and compared to existing) modelling frameworks in biology (e.g. cancer-related research, photosynthesis) in [73][74], in ecosystems (e.g. modelling vultures, mussels) in [17], biochemical networks, population dynamics, and linguistics, among others in [22][28].

Along with the practical applications, software simulators of P systems were developed, whether for generally sequential CPUs, or for massively parallel *graphics processing units* (in short, GPUs). Early simulators used *ad hoc* formats and designs, many of which can be found in [25]. A move to create a standard and unify these *ad hoc* simulators and formats birthed the programming language known as *P-Lingua*, which includes related tools (e.g. parsing, compiler) in [26]. To provide a generalized user interface for the numerous P system variants, the *Membrane Computing Simulator* (in short, MeCoSim) was developed, running as an upper layer (above P-lingua) to further support virtual experiments [82]. GPUs, due to their multi-level parallelism, were used to accelerate P system simulations. Early GPU simulations of P systems include [20], with a recent survey in [57].

Chapter 4

Spiking neural P systems (SNP systems)

In this chapter we introduce the specific P system which will be the focus of the dissertation: spiking neural P systems. Computing power and efficiency of spiking neural P systems are provided, which includes their various mathematically or biologically inspired variants. Applications of spiking neural P systems, both of theoretical and practical interest, are also given.

4.1 Introduction

In [80], the brain is identified as a "gold mine" of nature and bio-inspired (and hence, membrane computing) computations. The brain is an efficient and remarkable parallel and distributed "computer", capable of language and image processing, performing heuristics, and more, all while consuming 10 to 20 Watts of energy. Unless the organism (e.g. a human) is damaged or dies, the brain is never powered down or turned off. Such features make the brain a computer unlike any we have produced so far *in silico* (the brain in general does not also seem to share the problems of this technology). The adult human brain for example, contains approximately 10^{10} neurons where each neuron is connected to thousands of other neurons [42][44][54].

The biological neuron[1] is a very specialized type of cell consisting of the *soma* (the cell body), the *axon*, and the filamentous *dendrites* surrounding the soma. The dendrites are the "input lines" while the axon is the "output line" of the neuron, and the axon forms *synapses* or connections to other dendrites of other neurons.

Since the first generation of neural networks were introduced in the 1940s and 1950s (see [50][56][93], the newest generation, that of *spiking neurons*, has recently been introduced (see [34][53][54]) and has generated tremendous research interest. A good survey of the neural network generations is in [35]. With the identification of the spiking neuron generation, together with the solid mathematical foundations of membrane computing[2] *spiking neural P systems* (in short, SNP systems) were introduced in late 2005 and published in 2006 [44]. In spiking neurons and in

[1] A list of online and open access illustrations of biological neurons (e.g. structure, components) is in Appendix A.

[2] The theory of which is largely based on formal language and automata theory, and many other theoretical computer science areas.

SNP systems, the indistinct signals known as *spikes* used by biological neurons do not encode the information. Instead, information is derived for example from the time difference between two spikes, or the number of spikes sent (received) during a fixed interval. Time therefore is an information support in spiking neurons, and not simply a background for performing computations.

An SNP system can be thought of as a network of *spike processors*, processing spike objects and sending them to other neurons. Essentially, SNP systems can be represented by directed graphs where nodes are mono-membrane neurons (often drawn as ovals) and edges between neurons are synapses. Spikes (represented as the object a in the singleton alphabet $\{a\}$) are sent from one neuron to another using their synapses.

4.2 SNP systems syntax and semantics

A *spiking neural P system* of degree $m \geq 1$, is a construct of the form $\Pi = (\{a\}, \sigma_1, \ldots, \sigma_m, syn, in, out)$ where:

- $\{a\}$ is the singleton alphabet (a is called *spike*);
- $\sigma_1, \ldots, \sigma_m$ are *neurons* of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the *initial number of spikes* inside σ_i;
  - R_i is a finite *set of rules* of the general form: $E/a^c \rightarrow a^p; d$, where E is a regular expression over $\{a\}$, $c \geq 1$, with $p, d \geq 0$, and $c \geq p$; if $p = 0$, then $d = 0$ and $L(E) = \{a^c\}$;
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses*);
- $in, out \in \{1, \ldots, m\}$ indicate the *input* and *output* neurons, respectively.

A rule $E/a^c \rightarrow a^p; d$ in neuron σ_i (we also say neuron i or simply σ_i if there is no confusion) is called a *spiking rule* if $p \geq 1$. If $p = 0$, then $d = 0$ and $L(E) = \{a^c\}$, so that the rule is written simply as $a^c \rightarrow \lambda$, known as a *forgetting rule*. If a spiking rule has $L(E) = \{a^c\}$, we simply write it as $a^c \rightarrow a^p; d$. The systems from the original paper in [44], with rules of the form $E/a^c \rightarrow a; d$ and $a^c \rightarrow \lambda$, are referred to as *standard* systems with *standard rules*. The systems using *extended rules* (i.e. $p \geq 1$) are referred to as SNP systems with extended rules here as well as in [43][78][80].

The idea of the rule application is that a neuron contains a multiset of spikes over $\{a\}$. A regular "checking set" in the form of a regular expression E found in each rule is a condition that must be satisfied before a rule can be applied. More formally, if σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$, then the rule $E/a^c \rightarrow a^p; d \in R_i$ with $p \geq 1$, is enabled and can be applied. Rule application means consuming c spikes, so only $k - c$ spikes remain in σ_i. The neuron produces p spikes after d time units, to every σ_j where $(i, j) \in syn$, i.e. every neuron with a synapse from σ_i. We will use the terms "fire", "firing", or "spiking" to refer to the exit of spikes from a neuron after the neuron applies a spiking rule. If $d = 0$ then the p spikes arrive at the same time as rule application. If $d \geq 1$ and the time of rule application is t, then during the time sequence $t, t+1, \ldots, t+d-1$ the neuron is *closed*.

If a neuron is closed, it cannot receive spikes, and all spikes sent to it are lost. Starting at times $t + d$ and $t + d + 1$, the neuron becomes *open* (i.e., can receive spikes), and can apply rules again, respectively. Applying a forgetting rule means consuming spikes but producing no spikes. Note that a forgetting rule is never delayed since $d = 0$.

SNP systems operate under a global clock, i.e. they are *synchronous*: at every step, every neuron that can apply a rule must do so. It is possible that at least two rules $E_1/a^{c_1} \to a^{p_1}; d_1$ and $E_2/a^{c_2} \to a^{p_2}; d_2$, with $L(E_1) \cap L(E_2) \neq \emptyset$, can be applied at the same step. The system *nondeterministically* chooses exactly one rule to apply. The system is *globally parallel* (each neuron can apply a rule) but is *locally sequential* (a neuron can apply at most one rule).

A *configuration* or state of the system at time t can be described by $C_t = \langle r_1/t_1, \ldots, r_m/t_m \rangle$ for $1 \leq i \leq m$: Neuron i contains $r_i \geq 0$ spikes and it will open after $t_i \geq 0$ time steps. The initial configuration of the system is therefore $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$, where all neurons are initially open. Rule application provides us a *transition* from one configuration to another. A *computation* is any (finite or infinite) sequence of transitions, starting from a C_0. A halting computation is reached when all neurons are open and no rule can be applied.

Given neuron σ_i we define the set of neuron labels which has σ_i as their *presynaptic*[3] neuron as $pres(i)$, i.e. $pres(i) = \{j | (i, j) \in syn\}$. Similarly, we denote the set of neuron labels which has σ_i as their *postsynaptic* neuron as $pos(i) = \{j | (j, i) \in syn\}$. For the input and output neurons we have $pos(in) = \emptyset$, and $pres(\sigma_{out}) = \emptyset$, respectively. Essentially, $|pres(i)|$ and $|pos(i)|$ are the out and in-degrees of σ_i, respectively.

The output of a computation is defined in several ways in literature. Standard SNP systems as in [44] can generate sets of numbers as follows: the time steps when the first two spikes of σ_{out} are labeled as t_1 and t_2 and their difference, i.e. $t_2 - t_1 = n$, is said to be computed. This way of generating numbers usually applies to halting computations, although it can be applied to non-halting ones where the step for the third spike and beyond are ignored for example.

In accepting numbers, a number of spikes is stored in a specified neuron and the number is said to be accepted or computed if computation halts. Another way is to introduce two spikes into σ_{in} at steps t_1 and t_2, where $t_2 - t_1 = n$ is the number to be accepted. This way of introducing an input (producing an output, resp.) can be generalized as a *spike train*, e.g. to encode k natural numbers $n_1, n_2, \ldots, n_k$, we use as encoding the binary sequence $z = 0^b 10^{n_1} 10^{n_2} 1 \ldots 10^{n_k} 10^g$ for some $b, g \geq 0$. This encoding means that the input neuron receives (output neuron produces, resp.) a spike at the step corresponding to a 1 from the string z. The sets of numbers generated (accepted, resp.) by an SNP system Π in this way is denoted as $N_2(\Pi)$ (as $N_{2,acc}(\Pi)$, resp.). The subscript 2 indicates that we consider the first two spikes from the output neuron (into the input neuron, resp.), and *acc* denotes accepting. Next, we provide a simple example to demonstrate the SNP systems syntax and semantics discussed above.

[3]The terms presynaptic and postsynaptic neurons are taken from neuroscience.

Example 4.2.1. *Let* Π *(illustrated in Figure 4.1) be an SNP system generating the set* $\mathbb{N}^+ - \{1\}$ *constructed as follows:* $\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, 3)$*, where* $\sigma_1 = (2, R_1), \sigma_2 = (1, R_2), \sigma_3 = (3, R_3)$, $R_1 = \{a^2/a \to a, a \to \lambda\}, R_2 = \{a \to a, a \to a; 1\}, R_3 = \{a^3 \to a, a \to a; 1, a^2 \to \lambda\}$*,* $syn = \{(1,2),(2,1),(1,3),(2,)\}$.

As a convention, if the delay $d = 0$ we omit it from writing. Similar to automata illustrations, the input neuron (not given in Example 4.2.1) is drawn with an edge coming from the environment, while the output neuron is drawn with an edge going to the environment. In most of the examples and proofs in this work, computations by SNP systems will be described as follows, and illustrated as in Figure 4.1 instead of being formally defined, for the purpose of brevity and understandability.

The initial configuration of Π in Example 4.2.1 is $C_0 = \langle 2/0, 1/0, 3/0\rangle$, and computation is as follows: At step t_1, σ_1 with its two spikes applies its rule $a^2/a \to a$ (consuming one spike), σ_2 nondeterministically decides which among its two rules it will apply, and σ_3 having three spikes applies its rule $a^3 \to a$ (consuming all three spikes). Neuron σ_1 sends one spike to σ_2 and σ_3, σ_3 sends one spike to the environment (still at step t_1). If σ_2 applies its rule without delay, then it immediately sends one spike to σ_1 and σ_3 at t_1 and we have the configuration $C_1 = \langle 2/0, 1/0, 2/0\rangle$ (the net effect, after adding the spikes received by each neuron from other neurons). Otherwise we have $C_1' = \langle 1/0, 0/1, 1/0\rangle$ (again, the net effect) when σ_2 becomes closed.

It can be observed that as long as σ_2 applies its rule without delay, σ_2 keeps exchanging a spike with σ_1, while both neurons supply a total of two spikes to σ_3. As long as σ_3 receives two spikes, it must apply its forgetting rule $a^2 \to \lambda$ to remove the two spikes and produce no spike. However, once σ_2 applies its rule with a delay, σ_2 consumes its spike and becomes closed so that the spike from σ_1 is lost. In the next step (let us label this t_n), σ_1 applies its rule $a \to \lambda$ and then receives the delayed spike from σ_2 (which is now open). From the previous step σ_3 has one spike from σ_1 so it applies its rule $a \to a; 1$ and becomes closed. Hence, the spike sent by σ_2 to σ_3 at t_n is lost.

At t_{n+1}, σ_3 becomes open and spikes for the second and final time, after which Π halts. The output is $t_{n+1} - t_1 = n$, for $n \geq 2$, since at least two steps separate the first and second spike of σ_{out}. Therefore we have $N_2(\Pi) = \mathbb{N}^+ - \{1\}$.

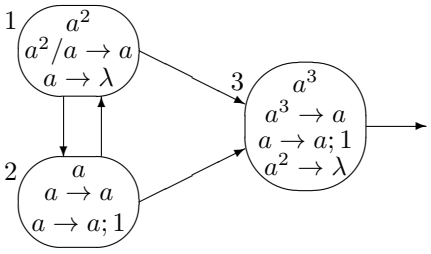

Figure 4.1: A simple example of an SNP system generating the set $\mathbb{N}^+ - \{1\}$.

4.3 Computing power

Universality for standard SNP systems, as well as boundaries for nonuniversality, were already established in the seminal paper in [44] for generating or accepting numbers. In [44] however, many open problems related to the parameters (e.g. number of neurons, rules in a neuron, types of rules) in the (non)universality results were given, including other input or output encodings. The standard SNP system originally from [44] included neurons with spiking rules that can have complex regular expressions, rules with delays, and forgetting rules. A series of papers which proved universality and constructed under a *normal form* (a set of simplifying requirements) such as restrictions on regular expressions of rules, removing delays or forgetting rules followed, e.g. [30][40], with the most recent being [66]. In [40] for example, SNP systems can be universal with the following normal form: rules are without delays; without using forgetting rules; regular expressions are simple. In [30] it was shown that universality is achieved with more restrictions: without delays and forgetting rules; without delays and rather simple regular expressions. Other normal forms in [29] include bounding the fan-out (outdegree) to two, combined with the removal of other features, the computing power of SNP systems is still maintained. Many of these works, and more, on the computing power of SNP systems, provide evidence to the robustness of the SNP systems model: removal or restriction of many features do not decrease the computing power of the model.

SNP systems often use parallel, nondeterministic, or synchronous features in their computations (common in membrane computing). These features are powerful "ingredients" for achieving universality, and many works (biologically or mathematically motivated) have investigated restrictions by removing at least one of these features and considering boundaries for (non)universality, e.g. deterministic SNP systems have been considered since [44]; asynchronous SNP systems as in [18][19][39][89]; sequential SNP systems as in [39][41][47][90]. Further details on the asynchronous or sequential modes of operation of SNP systems will be given in succeeding chapters of this work.

The use of extended rules allow for more compact systems in terms of smaller neuron count, due to the ability to produce more than one spike each step as in [21] and [78]. Computability results have also been investigated on SNP systems depending on the type of regular expressions in the rules as in [18] and [89]. An active line of research in membrane computing and in many other models of computation in general, is searching for small universal systems. In [78] for example, a universal SNP system using extended rules generating numbers is constructed having only 50 neurons, while 76 neurons are needed when using standard rules only. These numbers have since then been decreased, e.g. in [68][103]. A boundary between universality and nonuniversality is four and three neurons, respectively, using extended rules given in [61]. SNP systems universality, whether using standard or extended rules, have also been considered under language generation, e.g. in [21]. From finite strings and languages, computability in the processing of infinite sequences as in [27][75][77] have also been considered. We do not provide further information on these systems, and more information is found in their corresponding citations and references therein.

4.4 Other bio-inspired variants

In SNP system literature, many biologically inspired features have been introduced for computing use, producing many SNP system variants. The following are some of the more investigated variants in SNP systems literature.

SNP systems with weighted synapses, introduced in [96] and investigated in other works with a recent one in [103]. The biological motivation for adding weights to synapses is that biological neurons can have more than one synapse between each other. The weight of a synapse in an SNP system is inspired by the number of synapses between neurons in a biological neural network. Standard SNP systems are essentially SNP systems with weights where all synapse weights are equal to 1. Synapse weights function in a similar manner as extended rules, since the synapse weight acts as a "spike multiplier". With this spike multiplication, fewer neurons are required for achieving universality [96], or the system can be of a simpler form, i.e. a more restrictive normal form [103].

SNP systems with astrocytes were first introduced in [6], with varying biological motivations in succeeding works in [76][70]. In neuroscience, astrocytes are support structures: they can repair, feed, inhibit, or excite neurons. Universality results are obtained: in the first work in [6], astrocytes had a inhibitory and excitatory motivation, used to simulate Boolean gates; in [76], a restricted normal form than the systems in [6] are investigated but only for the inhibiting function; in [70], a more restricted normal form is provided for inhibitory and excitatory astrocytes. SNP systems with astrocytes from these work involve a second type of cell which is the astrocyte. An astrocyte as_i has a threshold t_i, and controls at least one synapse. The excitatory function allows spikes along synapses controlled by as_i to continue to their destination neurons, as long as the sum of the spikes do not exceed t_i. If the sum exceeds t_i, as_i performs an inhibitory function where all spikes along the synapses it controls are lost. If the sum is equal to t_i, as_i nondeterministically chooses an inhibitory or excitatory function.

SNP systems with anti-spikes were introduced in [65], and further investigated in several other works. In these systems, a second object in the alphabet is added, the anti-spike symbole $\overline{a}$. Anti-spikes, in part inspired by anti-matter and inhibitory synapses, annihilate spikes when they encounter each other inside a neuron. This annihilation comes in the form of a rule $a\overline{a} \to \lambda$ which has the highest priority and takes no time to be applied. In [65] then, a neuron either has spikes or antispikes inside it but not both. Spiking rules are also modified so that they can also produce either spikes or anti-spikes. The universality results obtained were simpler (i.e. contained less parameters, e.g. number of neurons, rules) than previous universality results of standard SNP systems due to the annihilation rule priority.

SNP systems with neuron division and budding as in [69][95] are inspired by neural stem cells which can create new neurons. Neuron division and neuron budding rules are two new rule types, together with spiking and forgetting rules. Since standard SNP systems are essentially systems with neuron division and budding rules removed, universality has not been the focus of these systems.

Instead, computational efficiency is investigated, provided in the succeeding section.

SNP systems with rules on synapses were recently introduced in [91], with the bio-motivation of providing more focus on the processing function of the axon and synapse. In neuroscience for example, the neuron is not the only structure that has an effect on the spike, but rather the axon and synapses can also affect spikes. Universality results are obtained, where the systems are simpler than constructed universal and standard SNP systems.

Each of these variants maintain their own open problems in computability, whether generating (sets) of numbers or languages, under asynchronous or sequential mode of operation, varying normal forms, etc. Computational efficiency results of some these variants are provided in the next section, together with other variants.

4.5 Computing efficiency and applications

SNP systems have been used to solve computer science and **NP**-complete problems, e.g. sorting networks and Boolean gates in [45] where SNP systems simulate circuits in linear time, the `SAT` problem was solved in constant time in [46] using exponential pre-computed resources.

In [51] a family of semi-uniform and nondeterministic SNP systems using extended rules solved the `Subset Sum` problem in constant time, while in [52] a uniform family of systems applying rules in a maximally parallel manner solved the problem in polynomial time. In these family of solutions, a semi-uniform solution involves a construction of the system where the system parameters (e.g. number of neurons, rules) are dependent on a particular problem instance. A uniform solution is one where the system construction and parameters are independent of the problem instance. This idea is similar to families of circuits where each input problem size requires a specific circuit to solve the problem. In particular, it was proven in [51] that without maximal parallelism in rule application, i.e. rules are only applied in the (standard) locally sequential way of at most one rule per neuron, then **NP**-complete problems cannot be solved in polynomial time.

A uniform solution solution to the `Subset Sum` problem was also given in [51] but the solution (i.e. the system) requires maximal parallelism. The solution was improved in [52], where the `SAT` problem was also solved, without using maximal parallelism and extended rules but still using exponential number of neurons. Smaller systems using maximal parallelism and extended rules are given in [62].

Since an **NP**-complete problem cannot be solved in polynomial time unless $\mathbf{P} = \mathbf{NP}$, other ideas to improve the efficiency of SNP systems were introduced. Aside from the above mentioned ideas, e.g. maximal parallelism or exponentially many pre-computed neurons, a bio-inspired idea was the use of neural stem cells to produce new neurons. This is the reason why the variant known as SNP systems with neuron division and budding were introduced. The system starts with a polynomial number of neurons with respect to the problem instance size of the `SAT` problem. Using the additional neuron division and budding rules, an exponential number of neurons are produced to solve the problem in

a uniform way and in polynomial time.

Applications to practical and engineering problems, using or inspired by SNP systems, are relatively recent, e.g. in [24][97][98] and references therein. Much investigation remains to be done in order to apply SNP systems and their theory to such practical problems. There also exist several software simulators for SNP systems, in connection with the software for other P systems mentioned in the previous chapter. The variants in Section 4.4 together with other execution modes (e.g. asynchronous, maximally parallel), and many others besides these, are included in the P-Lingua library as in [55]. Preliminary GPU simulators for restricted standard SNP systems are also given in [7] [8].

Part II

Body of Work

Chapter 5

Notes on delays in SNP systems

In many SNP systems solutions, i.e. system constructions, adding delays in rules can be a useful "programming" feature, where applying a delay produces a different computation than when it is not applied [19][44][45][52]. Although SNP systems can achieve universality without using delays as in [66], delays can be useful in order to distinguish certain computations from others. For example, using delays is useful when certain features (e.g. forgetting rules, extended rules, synchronous operation) of the system is restricted or removed. The constructions provided in this chapter are intended to be support SNP systems, in the sense of *routing modules*[1], which can be used in constructing larger SNP systems for any purpose. We provide restrictions so that the output of an SNP system (module) Π with standard rules having delays can be produced by an SNP system (module) $\overline{\Pi}$ with standards rules having no delays. The constructions presented here for example, can be used whether SNP systems constructed with our routing modules are used to generate or accept (sets) of numbers or languages.

5.1 Introduction

Consider the SNP system with delay Π_0 shown in Figure 5.1, where the single spike will be *routed* (i.e. transferred) eventually to the output neuron. Only neuron σ_1 has one spike initially, and only σ_2 has a rule with a delay $d = 2$, thus we have $C_0 = \langle 1/0, 0/0, 0/0, 0\rangle$. In this chapter we have a minor modification of the convention in Section 4.2 representing the configuration: the last (rightmost) element of the vector C_k represents the number of spikes in the environment. At the next step, σ_1 must use its rule (it has at least one spike, and $a \in L(a^+)$), then consume one spike and send one spike immediately to σ_2. We thus have $C_1 = \langle 0/0, 1/0, 0/0, 0\rangle$. At step 2, σ_2 consumes its spike and closes for 2 time steps, so $C_2 = \langle 0/0, 0/2, 0/0, 0\rangle$, while at step 3 we have $C_3 = \langle\ 0/0, 0/1, 0/0, 0\rangle$. At step 4, σ_2 opens and sends one spike to σ_3, so $C_4 = \langle 0/0, 0/0, 1/0, 0\rangle$. Finally, at step 5 the output neuron sends one spike to the environment, Π_0 halts and we have $C_5 = \langle 0/0, 0/0, 0/0, 1\rangle$.

SNP systems where each σ_i has exactly one rule (i.e. $|R_i| = 1$) are called *simple*, while the

[1] So as not to be confused with the more general SNP modules later in a succeeding chapter.

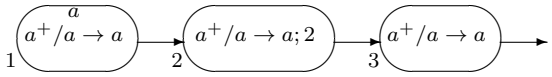

Figure 5.1: SNP system with delay Π_0.

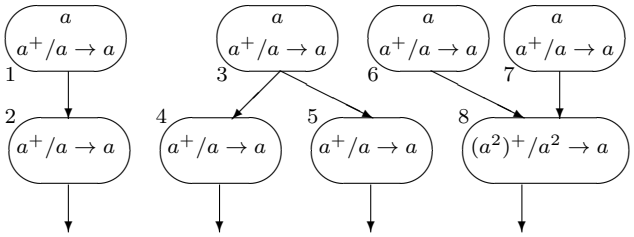

Figure 5.2: Routing modules (from left to right): sequential, split, and join.

systems that have the same set of rules are called *homogeneous* [100]. In this chapter, we only consider SNP systems that have rules of the restricted form $(a^k)^+/a^k \to a; d$ where k is a positive integer, and refer to them as *semi-homogeneous*. We only consider SNP systems Π and $\overline{\Pi}$ that are simple and semi-homogeneous, i.e. every σ_i in either Π or $\overline{\Pi}$ has the rule set $R_i = \{(a^k)^+/a^k \to a; d\}$. Both Π and $\overline{\Pi}$ have *initial neuron(s)*, which are the only neurons with a single spike in their initial configurations (as in Figure 5.1). We make no restrictions on the values of the delay d in a rule. We then route or move the single spike in the initial neuron through the system, towards the output neuron, and eventually to the environment.

Spikes are routed via *paths*. There exists a path from σ_1 to σ_k if given neurons $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{k-1}, \sigma_k$, we have synapses $\{(1,2),(2,3),\ldots,(k-1,k)\} \subseteq syn$. Using paths, we can have the following routing modules (referring to Figure 5.2): *sequential* routing where, given at least two neurons σ_1 and σ_2, σ_2 spikes only after σ_1 spikes, a path exists from σ_1 to σ_2, and in general for a sequential path from σ_1 to σ_k, each consecutively connected neuron σ_i has $pres(i) = \{i+1\}, i \in \{1,2,\ldots,k\}$; *split* routing, where the initial neuron σ_3 is the only presynaptic neuron of at least two succeeding neurons σ_4 and σ_5, with $(3,4),(3,5) \in syn$; *join* routing, where at least two initial neurons σ_6, σ_7 are the only presynaptic neurons of a single succeeding neuron σ_8, so that $(6,8),(7,8) \in syn$, and σ_8 produces a spike only after accumulating spikes from σ_7 and σ_8.

Notice that an iteration routing can be formed by using one of the previous modules, e.g. if we add synapse $(2,1)$ in Figure 5.2, then a loop is formed and spikes are iteratively produced. Also notice that if there exists a sequential path from σ_i (with delay d_i) to σ_j (with delay d_j) with $d_i < d_j$ and the number of spikes of the initial neuron σ_i in C_0 is $n_i > 1$, it is possible for some spikes to be lost during computation. The reason is that it is possible for σ_j to still be closed when spikes from σ_i arrive. We avoid lost spikes by considering SNP systems where the initial neuron has only one spike, since the intention for these constructions is to be used as a part of a larger system and

removing or losing spikes is handled elsewhere. The initial neuron can then have another spike at a step greater than the time it takes for the neuron with largest delay in the path to open, in order not to lose any spikes. We say in this chapter that a $\overline{\Pi}$ *route simulates*[2] a Π if two requirements are satisfied: (R1) the halting step of Π is the same halting step of $\overline{\Pi}$, and (R2) the number of spikes in the environment of Π when Π halts is equal to the number of spikes in the environment of $\overline{\Pi}$ when $\overline{\Pi}$ halts. Requirement (R1) allows us to use our constructions in SNP systems that assign a number to the time difference between spikes, while (R2) is for SNP systems where strings are considered and associated with spike multiplicity.

5.2 Results

In the succeeding results we have the following writing convention: an SNP system Π with delay has neuron labels $1i$, while an SNP system $\overline{\Pi}$ without delay has neuron labels $2j, i, j \in \mathbb{N}$.

Lemma 5.2.1. *Let Π_1 be a sequential routing module and a simple, semi-homogeneous SNP system with delays. Then we can construct a sequential routing module and a simple, semi-homogeneous SNP system without delays $\overline{\Pi}_1$ that route simulates Π_1.*

Proof. We refer to Figure 5.3 for illustrations. The additional d neurons, immediately after initial neuron σ_{21} in $\overline{\Pi}_1$, are used to multiply the single spike from σ_{21}. The additional neurons then send one spike each to $\sigma_{22\text{-}d}$. Neuron $\sigma_{22\text{-}d}$ accumulates $d-1$ spikes, and consumes these, one spike at a time. Each step, $\sigma_{22\text{-}d}$ sends one spike to σ_{23}. This consumption of one spike every step creates a delay of $d-1$ steps. Due to the regular expression of the rule in σ_{23}, the neuron will have to accumulate $d-1$ spikes before the rule is used. Once σ_{23} accumulates $d-1$ spikes, it immediately sends one spike to the environment. This spiking and halting occurs at time $t+d+1$ for both Π and $\overline{\Pi}$, where t is the time when σ_{11} and σ_{21} spike, thus satisfying (R1). Clearly we satisfy (R2) also, as the number of spikes produced by both output neurons of Π_1 and $\overline{\Pi}_1$ are equal.

We can repeatedly apply the previous construction if there exists more than one neuron with a (rule having a) delay in a sequential path as seen in Figure 5.4. It can be easily shown that if there exists σ_i without delay in a sequential path between σ_{11} and σ_{12}, the time to halt for both Π_1 and $\overline{\Pi}_1$ still coincide. In particular, every additional σ_i having a rule without a delay adds one time step to the halting time of both Π_1 and $\overline{\Pi}_2$. Both (R1) and (R2) are still satisfied. □

A sample computation of Π_1 and $\overline{\Pi}_1$ is shown in Table 5.1. For sample computations of Π_2 and $\overline{\Pi}_2$ we refer to Table 5.2. From Lemma 5.2.1 we have the following observation.

Observation 5.2.1. *If Π_1 has more than one neuron with a delay in a rule, the total additional neurons in $\overline{\Pi}_1$ is $\sum_{i=1}^{m} d_i$ where d_i is the delay of the rule in σ_i.*

[2]To remove the confusion of using the term "simulation" as it used in computability theory and in succeeding chapters in this work.

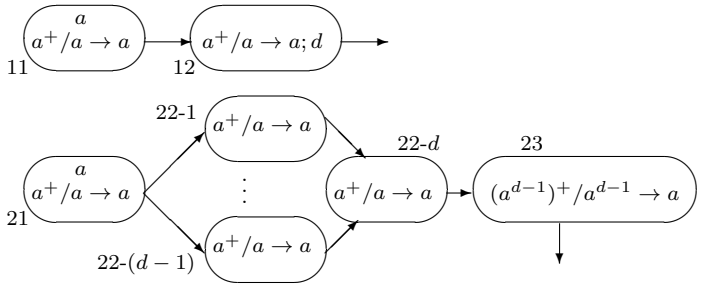

Figure 5.3: Sequential routing: Π_1 (top) with delay d, and $\overline{\Pi}_1$ (bottom) route simulating Π_1.

Steps	Π_1	$\overline{\Pi}_1$
t_0	$\langle 1/0, 0/0, 0\rangle$	$\langle 1, 0, 0, 0, 0, 0\rangle$
t_1	$\langle 0/0, 1/0, 0\rangle$	$\langle 0, 1, 1, 0, 0, 0\rangle$
t_2	$\langle 0/0, 0/3, 0\rangle$	$\langle 0, 0, 0, 2, 0, 0\rangle$
t_3	$\langle 0/0, 0/2, 0\rangle$	$\langle 0, 0, 0, 0, 1, 0\rangle$
t_4	$\langle 0/0, 0/1, 0\rangle$	$\langle 0, 0, 0, 0, 2, 0\rangle$
t_5	$\langle 0/0, 0/0, 1\rangle$	$\langle 0, 0, 0, 0, 0, 1\rangle$

Table 5.1: Sample computations of Π_1 and $\overline{\Pi}_1$, $d = 3$.

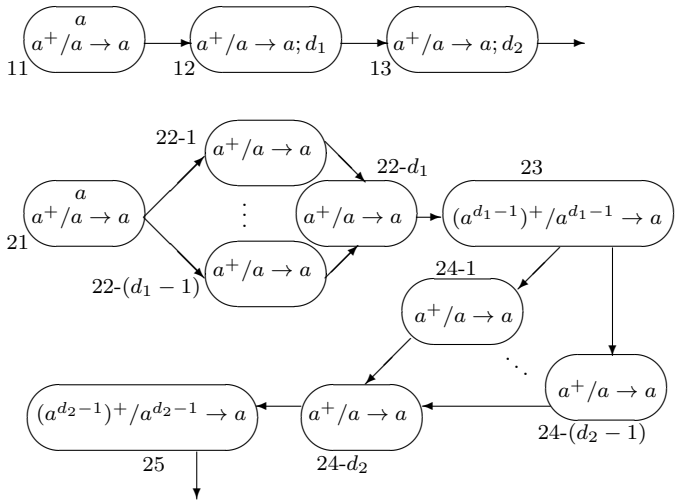

Figure 5.4: Sequential routing with multiple delays: Π_2 (top) with delays d_1 and d_2, and $\overline{\Pi}_2$ (bottom) route simulating Π_2.

Steps	Π_2	$\overline{\Pi}_2$
t_0	$\langle 1/0,\ 0/0,\ 0/0,\ 0\rangle$	$\langle 1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\rangle$
t_1	$\langle 0/0,\ 1/0,\ 0/0,\ 0\rangle$	$\langle 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\rangle$
t_2	$\langle 0/0,\ 0/2,\ 0/0,\ 0\rangle$	$\langle 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\rangle$
t_3	$\langle 0/0,\ 0/1,\ 0/0,\ 0\rangle$	$\langle 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0\rangle$
t_4	$\langle 0/0,\ 0/0,\ 1/0,\ 0\rangle$	$\langle 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 0\rangle$
t_5	$\langle 0/0,\ 0/0,\ 0/3,\ 0\rangle$	$\langle 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 2,\ 0,\ 0\rangle$
t_6	$\langle 0/0,\ 0/0,\ 0/2,\ 0\rangle$	$\langle 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0\rangle$
t_7	$\langle 0/0,\ 0/0,\ 0/1,\ 0\rangle$	$\langle 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 2,\ 0\rangle$
t_8	$\langle 0/0,\ 0/0,\ 0/0,\ 1\rangle$	$\langle 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1\rangle$

Table 5.2: Sample computations of Π_2 and $\overline{\Pi}_2$, $d_1 = 2$, $d_2 = 3$.

Observation 5.2.2. *A Π_1 with iteration can be route simulated by a $\overline{\Pi}_1$.*

The construction for $\overline{\Pi}_3$ uses the construction idea in Observation 5.2.1 i.e. the neuron with a delay d in Π_3 is replaced with d additional neurons in $\overline{\Pi}_3$. In Figure 5.5 an infinite loop is created: a spike starts at σ_{11} and it uses its rule at step t so that the spike is sent to σ_{12} at step $t + d$, then σ_{12} immediately sends a spike back to σ_{11} (and the environment) at step $t + d + 1$, and so on and so forth. Similarly, σ_{21} sends a spike to neurons $\sigma_{22\text{-}1}$ to $\sigma_{22\text{-}(d-1)}$ at step t. At step $t + 1$, $\sigma_{22\text{-}d}$ accumulates $d - 1$ spikes from the $d - 1$ neurons from the previous step. The spikes in $\sigma_{22\text{-}d}$ are consumed and a spike is sent each step to σ_{23}. At step $t + d$, σ_{23} accumulates $d - 1$ spikes so that it sends one spike back to σ_{21} and at the environment at step $t + d + 1$, coinciding with the step of spiking of σ_{12}. Thus, (R1) and (R2) are satisfied.

Observation 5.2.2 for iteration routing makes use of the construction used in Lemma 5.2.1 for sequential routing. This construction will again be used for the join and split routings as follows. Additionally we have the following observation.

Observation 5.2.3. *If the initial neuron of Π_1 has a delay and its halting time is $t + d$, we add a new initial neuron $\sigma_{1'}$ in Π_1 with $(1', 1) \in syn$ so that Π_1 halts at time $t + d + 1$. We then add a new initial neuron similarly to $\overline{\Pi}_1$ and modify its syn (following Lemma 5.2.1 construction) so that $\overline{\Pi}_1$ halts at $t + d + 1$ instead, route simulating Π.*

For example, if σ_{11} has a delay instead of σ_{12} in Π_3, we add a new initial neuron to $\sigma_{11'}$ and a new synapse $(11', 11) \in syn$ in Π. For $\overline{\Pi}_3$, we modify it as follows: add a new initial neuron $\sigma_{21'}$ and σ_{21} is replaced with d parallel neurons previously found in $\sigma_{22\text{-}i}, i \in \{1, 2, \ldots, d-1\}$. The synapse set of $\overline{\Pi}_3$ is changed to $syn = \{(21', 21\text{-}1), \ldots, (21', 21\text{-}(d-1))\} \cup \{(21\text{-}1, 21\text{-}d), \ldots, (21\text{-}(d-1), 21\text{-}d)\} \cup \{(21\text{-}d, 22),\ (22, 21\text{-}1),\ \ldots,\ (22, 21\text{-}(d-1))\}$, we remove σ_{23} and have σ_{22} as the output neuron instead. Both Π_3 and $\overline{\Pi}_3$ halt at the same time at $t + d + 2$.

Lemma 5.2.2. *Let Π_4 be a join routing module and a simple, semi-homogeneous SNP system with delays. Then we can construct a join routing module and a simple, semi-homogeneous SNP system*

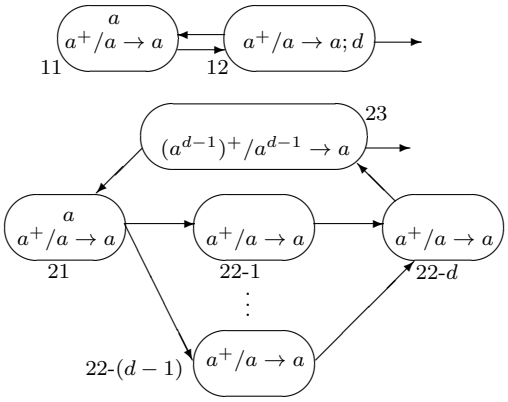

Figure 5.5: Iteration routing: Π_3 (top) has delay d route simulated by $\overline{\Pi}_3$ (bottom).

Steps	Π_4	$\overline{\Pi}_4$
t_0	$\langle 1/0, 1/0, 0/0, 0\rangle$	$\langle 1, 1, 0, 0, 0, 0, 0\rangle$
t_1	$\langle 0/0, 0/0, 2/0, 0\rangle$	$\langle 0, 0, 2, 2, 0, 0, 0\rangle$
t_2	$\langle 0/0, 0/0, 0/3, 0\rangle$	$\langle 0, 0, 0, 0, 4, 0, 0\rangle$
t_3	$\langle 0/0, 0/0, 0/2, 0\rangle$	$\langle 0, 0, 0, 0, 2, 1, 0\rangle$
t_4	$\langle 0/0, 0/0, 0/1, 0\rangle$	$\langle 0, 0, 0, 0, 0, 2, 0\rangle$
t_5	$\langle 0/0, 0/0, 0/0, 1\rangle$	$\langle 0, 0, 0, 0, 0, 0, 1\rangle$

Table 5.3: Sample computations of Π_4 and $\overline{\Pi}_4$, $d = 3$.

without delays $\overline{\Pi}_4$ *that route simulates* Π_4.

For Π_4 and $\overline{\Pi}_4$ we have as initial neurons σ_{11}, σ_{12} and σ_{21}, σ_{22} respectively, although an arbitrary number of input neurons for both systems can be used. Using the construction in Lemma 5.2.1, $\overline{\Pi}_4$ has d additional neurons corresponding to σ_{13} in Π. All initial neurons in $\overline{\Pi}_4$ have a synapse to all $\sigma_{23\text{-}i}, i \in \{1, 2, \ldots, d-1\}$. All neurons $\sigma_{23\text{-}i}, i \in \{1, 2, \ldots, d-1\}$ in turn each have a synapse to $\sigma_{23\text{-}d}$. Let time t be the step when the initial neurons spike. At t, neurons $\sigma_{23\text{-}1}$ to $\sigma_{23\text{-}(d-1)}$ have two spikes each, so that in total, $\overline{\Pi}_4$ at this time has $2(d-1)$ spikes. At the next step $t+1$, the $d-1$ neurons send two spikes each to $\sigma_{23\text{-}d}$ so that $\sigma_{23\text{-}d}$ accumulates $2(d-1)$ spikes. Since $\sigma_{23\text{-}d}$ consumes two spikes every time and it has $2(d-1)$ spikes, $\sigma_{23\text{-}d}$ will take $d-1$ steps to consume all of its $2(d-1)$ spikes. Every time $\sigma_{23\text{-}d}$ spikes, it sends only one spike to σ_{24}. At $t+d$, σ_{24} has accumulated $d-1$ spikes from $\sigma_{23\text{-}d}$ so that σ_{24} sends one spike to the environment and halts at $t+d+1$. This step coincides with the halting time of σ_{13} in Π_4, sending one spike to the environment. We therefore satisfy (R1) and (R2). □

A sample computation of Π_4 and $\overline{\Pi}_4$ is shown in Table 5.3.

Lemma 5.2.3. *Let* Π *be a split routing module and a simple, semi-homogeneous SNP system with delays. Then we can construct a split routing module and a simple, semi-homogeneous SNP system without delays* $\overline{\Pi}$ *that route simulates* Π.

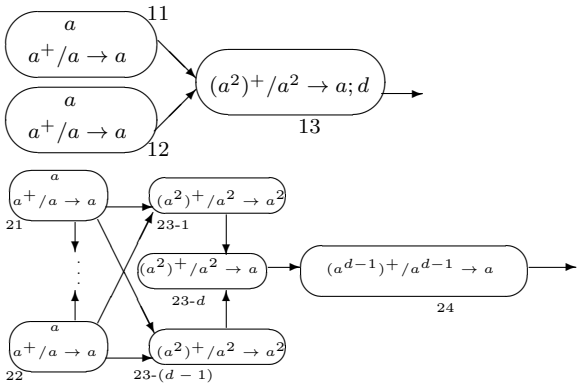

Figure 5.6: Join routing : Π_4 (top) has delay d route simulated by $\overline{\Pi}_4$ (bottom).

Proof. We refer back to the split routing in Figure 5.2. Notice that a split routing can be thought of as two separate paths, either from σ_3 to σ_4 or σ_3 to σ_5. We let t be the step that σ_3 spikes and modify the split routing in Figure 5.2 as follows: we add a new output neuron σ_o, and connect it from σ_4 and σ_5. Let

$$\Pi = (O, \sigma_3, \sigma_4, \sigma_5, \sigma_o, syn, o)$$

where $\sigma_3 = (1, a^+/a \to a)$, $\sigma_5 = (0, a^+/a \to a)$, $\sigma_4 = (0, a^+/a \to a; d)$, $syn = \{(3,4), (3,5), (4,o), (5,o)\}$. We arbitrarily chose σ_4 to have a delay instead of σ_5 in this case. Next we let

$$\overline{\Pi} = (O, \sigma_{3'}, \sigma_{4'\text{-}i}, \sigma_{5'}, \sigma_o, syn, o)$$

where $1 \leq i \leq d$, $\sigma_{3'} = (1, a^+/a \to a)$, $\sigma_{4'\text{-}1} = \ldots = \sigma_{4'\text{-}(d-1)} = \sigma_{5'} = (0, a^+/a \to a)$, $\sigma_{4'\text{-}d} = (0, (a^{d-1})^+/a^{d-1} \to a)$, $syn = \{(3', 4'\text{-}1), \ldots, (3', 4'\text{-}(d-1))\} \cup \{(3', 5'), (4'\text{-}1, 4'\text{-}d), \ldots, (4'\text{-}(d-1), 4'\text{-}d), (4'\text{-}d, o), (5, o)\}$.

Since σ_4 has delay d, we simply follow Lemma 5.2.1 and add d neurons in $\overline{\Pi}$ corresponding to σ_4. Let t be the step when initial neurons σ_3 and $\sigma_{3'}$ spike. The step that σ_o spikes the second time (the spike from σ_5 makes σ_o spike the first time) i.e. the halting step, is $t + d + 1$ and the environment receives two spikes in total. This step coincides with the halting step of $\overline{\Pi}$, which also sends two spikes to its environment. (R1) and (R2) are both satisfied for this case.

In the case where both σ_4 and σ_5 have delays d_4 and d_5 respectively, then following Lemma 5.2.1, $\overline{\Pi}$ has $d_4 + d_5$ additional neurons. For both systems, halting step is $t + d_{max} + 1$ where $d_{max} = max(d_4, d_5)$, and two spikes are sent to the environment, satisfying (R1) and (R2). □

We can now have the following Theorem, from Lemma 5.2.1, 5.2.2, and 5.2.3.

Theorem 5.2.1. *Let* Π *be a simple, semi-homogeneous SNP system with delays containing one or more of the following routing modules: sequential, join, split. Then there exists a simple, semi-homogeneous SNP system* $\overline{\Pi}$ *without delays that route simulates* Π.

Chapter 6

Notes on SNP systems and finite automata

SNP systems with standard rules have neurons that emit at most one spike (the pulse) each step, and have either an input or output neuron connected to the environment. In [77], SNP transducers were introduced, where both input and output neurons were used. More recently, SNP modules were introduced in [43] which generalize SNP transducers: extended rules are used (more than one spike can be emitted each step) and a set of input and output neurons can be used. In this chapter we continue relating SNP modules and finite automata, and in particular (i) we amend previous constructions for DFA and DFST simulations, (ii) improve the construction from three neurons down to one neuron, (iii) DFA with output are simulated, and (iv) we generate automatic sequences using results from (iii). Result (ii) is also the optimal result in terms of the number of neurons in a module.

6.1 Introduction

SNP systems with standard rules (as introduced in the seminal paper in [44]) have neurons that emit at most one spike each step, with either an input or output neuron connected to the environment, but not both. In [77], SNP systems were equipped with both an input and output neuron (known as *SNP transducers*) for handling infinite sequences. Furthermore, extended rules were introduced in [21] and [78], so that a neuron can produce more than one spike each step. The introduced *SNP modules* in [43] can then be seen as generalizations of SNP transducers: more than one spike can enter or leave the system, extended rules are used, and more than one neuron can function as input or output neuron.

The results in this chapter amend the problem introduced in the construction of [43], where SNP modules were used to simulate deterministic finite automata and state transducers. Our constructions also reduce the neurons for such SNP modules: from three neurons down to one. Our reduction relies on more involved superscripts, similar to some of the constructions in [61]. We also provide constructions for SNP modules simulating DFA with output. Establishing simulations between DFA with output and SNP modules, we are then able to use SNP modules to generate automatic

sequences. Such class of sequences contain, for example, a common and useful automatic sequence known as the Thue-Morse sequence. Automatic sequences also play important roles in many areas of mathematics (e.g. number theory) and computer science (e.g. automata theory). Aside from DFA with output, another way to generate automatic sequences is by iterating morphisms. We invite the interested reader to [3] for further theories and applications related to automatic sequences.

In this chapter, instead of computing numbers, a string over a finite arbitrary alphabet is associated with the output neuron. If σ_{out} produces i spikes in a step, we associate the symbol b_i to that step. In particular, the system (using rules in its output neuron) generates strings over $\Sigma = \{p_1, \ldots, p_m\}$, for every rule $r_\ell = E_\ell/a^{j_\ell} \to a^{p_\ell}; d_\ell, 1 \leq \ell \leq m$, in σ_{out}. From [21] we can have two cases: associating b_0 (when no spikes are produced) with a symbol, or as λ. In this work and as in [43], we only consider the latter.

A *spiking neural P module* (in short, an *SNP module*) of degree $m \geq 1$, is a construct of the form $\Pi = (\{a\}, \sigma_1, \ldots, \sigma_m, syn, N_{in}, N_{out})$ where

- $\{a\}$ is the singleton alphabet (a is called *spike*);
- $\sigma_1, \ldots, \sigma_m$ are *neurons* of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the *initial number of spikes* inside σ_i;
  - R_i is a finite *set of rules* of the general form: $E/a^c \to a^p$, where E is a regular expression over $\{a\}$, $c \geq 1$, and $p \geq 0$, with $c \geq p$; if $p = 0$, then $L(E) = \{a^c\}$
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses*);
- $N_{in}, N_{out} (\subseteq \{1, 2, \ldots, m\})$ indicate the *sets of input* and *output* neurons, respectively.

In [77], SNP transducers operated on strings over a binary alphabet as well considering b_0 as a symbol. SNP modules, first introduced in [43], are a special type of SNP systems with extended rules, and they generalize SNP transducers.

SNP modules behave in the usual way as SNP systems, except that spiking and forgetting rules now both contain no delays. In contrast to SNP systems, SNP modules have the following *distinguishing feature*: at each step, each input neuron $\sigma_i, i \in N_{in}$, takes as input *multiple copies* of a from the environment; Each output neuron $\sigma_o, o \in N_{out}$, produces p spikes to the environment, if a rule $E/a^c \to a^p$ is applied in σ_o; Note that $N_{in} \cap N_{out}$ is not necessarily empty.

6.2 DFA and DFST simulations

We briefly recall the constructions from Theorem 8 and 9 of [43] for SNP modules simulating DFAs and DFSTs. The constructions for both DFAs and DFSTs have a similar structure, which is shown in Figure 6.1. For neurons 1 and 2 in Figure 6.1, the spikes and rules for DFA and DFST simulation are equal, so the constructions only differ for the contents of neuron 3. Let $D = (Q, \Sigma, \delta, q_1, F)$ be a DFA, where $\Sigma = \{b_1, \ldots, b_m\}$, $Q = \{q_1, \ldots, q_n\}$. The construction for Theorem 8 of [43] for an SNP Module Π_D simulating D is as follows:

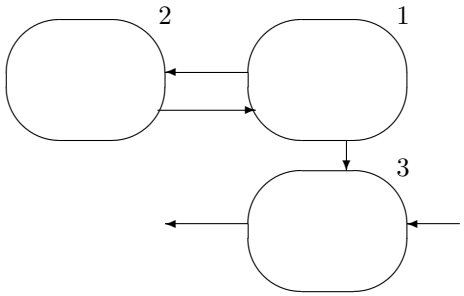

Figure 6.1: Structure of SNP modules from [43] simulating DFAs and DFSTs.

$$\Pi_D = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, \{3\}, \{3\}),$$

where

- $\sigma_1 = \sigma_2 = (n, \{a^n \to a^n\})$,
- $\sigma_3 = (n, \{a^{2n+i+k}/a^{2n+i+k-j} \to a^j | \delta(q_i, b_k) = q_j\})$,
- $syn = \{(1,2), (2,1), (1,3)\}$.

The structure for Π_D is shown in Figure 6.1. Note that $n, m \in \mathbb{N}$, are fixed numbers, and each state $q_i \in Q$ is represented as a^i spikes in σ_3, for $1 \leq i \leq n$. For each symbol $b_k \in \Sigma$, the representation is a^{n+k}. The operation of Π_D is as follows: σ_1 and σ_2 interchange a^n spikes at every step, while σ_1 also sends a^n spikes to σ_3.

Suppose that D is in state q_i and will receive input b_k, so that σ_3 of Π_D has a^i spikes and will receive a^{n+k} spikes. In the next step, σ_3 will collect a^n spikes from σ_1, a^{n+k} spikes from the environment, so that the total spikes in σ_3 is a^{2n+i+k}. A rule in σ_3 with $L(E) = \{a^{2n+i+k}\}$ is applied, and the rule consumes $2n + i + k - j$ spikes, therefore leaving only a^j spikes. A single state transition $\delta(q_i, b_k) = q_j$ is therefore simulated.

With a 1-step delay, Π_D receives a given input $w = b_{i_1}, \ldots, b_{i_r}$ in Σ^* and produces a sequence of states $z = q_{i_1}, \ldots, q_{i_r}$ (represented by $a^{i_1}, \ldots, a^{i_r}$) such that $\delta(q_{i_\ell}, b_{i_\ell}) = q_{i_{\ell+1}}$, for each $\ell = 1, \ldots, r$ where $q_{i_1} = q_1$. Then, w is accepted by D (i.e. $\delta(q_1, w) \in F$) iff $z = \Pi_D(w)$ ends with a state in F (i.e. $q_{i_r} \in F$). Let the language accepted by Π_D be defined as:

$$L(\Pi_D) = \{w \in \Sigma^* | \Pi_D(w) \in Q^* F\}.$$

Then, the following is Theorem 8 from [43]

Theorem 6.2.1. *(Ibarra et al [43]) Any regular language L can be expressed as $L = L(\Pi_D)$ for some SNP module Π_D.*

The simulation of DFSTs requires a slight modification of the DFA construction. Let $T =$

$(Q, \Sigma, \Delta, \delta', q_1, F)$ be a DFST, where $\Sigma = \{b_1, \ldots, b_k\}$, $\Delta = \{c_1, \ldots, c_t\}$, $Q = \{q_1, \ldots, q_n\}$. We construct the following SNP module simulating T:

$$\Pi_T = (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, \{3\}, \{3\}),$$

where:

- $\sigma_1 = \sigma_2 = (n, \{a^n \to a^n\})$,
- $\sigma_3 = (n, \{a^{2n+i+k+t}/a^{2n+i+k+t-j} \to a^{n+s} | \delta'(q_i, b_k) = (q_j, c_s)\})$,
- $syn = \{(1,2), (2,1), (1,3)\}$.

The structure for Π_T is shown in Figure 6.1. Note that $n, m, t \in \mathbb{N}$ are fixed numbers. For $1 \leq i \leq n, 1 \leq s \leq t, 1 \leq k \leq m$: each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented by a^i, a^{n+t+k}, and a^{n+s} spikes, respectively.

The operation of Π_T given an input $w \in \Sigma^*$ is in parallel to the operation of Π_D; the difference is that the former produces a $c_s \in \Delta$, while the latter produces a $q_i \in Q$. From the construction of Π_T and the claim in Theorem 6.2.1, the following is Theorem 9 from [43]:

Theorem 6.2.2. *(Ibarra et al[43]) Any finite transducer T can be simulated by some SNP module Π_T.*

The previous constructions from [43] on simulating DFAs and DFSTs have however, the following technical problem:

Suppose we are to simulate DFA D with at least two transitions, (1) $\delta(q_i, b_k) = q_j$, and (2) $\delta(q_{i'}, b_{k'}) = q_{j'}$. Let $j \neq j', i = k'$, and $k = i'$. The SNP module Π_D simulating D then has at least two rules in σ_3: $r_1 = a^{2n+i+k}/a^{2n+i+k-j} \to a^j$, simulating (1), and $r_2 = a^{2n+i'+k'}/a^{2n+i'+k'-j'} \to a^{j'}$ simulating (2).

Observe that $2n + i + k = 2n + i' + k'$, so that in σ_3, the regular expression for r_1 is exactly the regular expression for r_2. We therefore have a nondeterministic rule selection in σ_3. However, D being a DFA, transitions to two different states q_j and $q_{j'}$. Therefore, Π_D is a nondeterministic SNP module that can, at certain steps, incorrectly simulate the DFA D. This nondeterminism also occurs in the DFST simulation. An illustration of the problem is given in example 6.2.1.

Example 6.2.1. *We modify the 2-DFAO in Figure 2.1 into a DFA in Figure 6.2 as follows: Instead of $\Sigma = \{0, 1\}$, we have $\Sigma = \{1, 2\}$*

In Example 6.2.1 we maintain $n = m = 2$, however, the transitions are swapped, so in Figure 6.2 we have the following two (among four) transitions: $\delta(q_1, 2) = q_2$, and $\delta(q_2, 1) = q_1$. These two transitions cause the nondeterministic "problem" for the SNP module given in Figure 6.3. The problem concerns the simulation of the two previous transitions using rules $a^7/a^5 \to a^2$ and $a^7/a^6 \to a$ in σ_3, which can be nondeterministically applied: if σ_3 contains a^2 spikes and receives

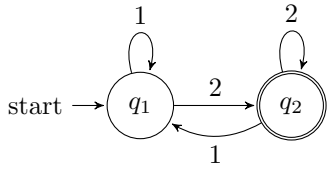

Figure 6.2: DFA with incorrect simulation by the SNP module in Figure 6.3.

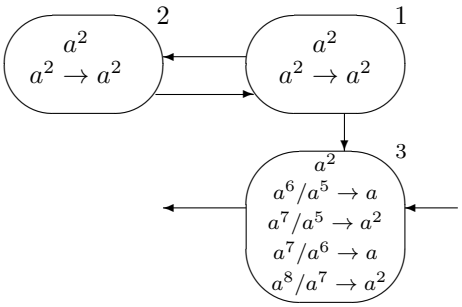

Figure 6.3: SNP module with incorrect simulation of the DFA in Figure 6.2.

a^3 from environment (representing input 1 for the DFA), at the next step σ_3 will have a^7 spikes, allowing the possibility of an incorrect simulation.

Next, we amend the problem and modify the constructions for simulating DFAs and DFSTs in SNP modules. Given a DFA D, we construct an SNP module Π'_D simulating D as follows:

$$\Pi'_D = (\{a\}, \sigma_1, syn, \{1\}, \{1\}),$$

where

- $\sigma_1 = (1, \{a^{k(2n+1)+i}/a^{k(2n+1)+i-j} \to a^j | \delta(q_i, b_k) = q_j\})$,
- $syn = \emptyset$.

We have Π_D containing only one neuron, which is both the input and output neuron. Again, $n, m \in \mathbb{N}$ are fixed numbers. Each state q_i is again represented as a^i spikes, for $1 \leq i \leq n$. Each symbol $b_k \in \Sigma$ is now represented as $a^{k(2n+1)}$ spikes. The operation of Π'_D is as follows: neuron 1 starts with a^1 spike, representing q_1 in D. Suppose that D is in some state q_i, receives input b_k, and transitions to q_j in the next step. We then have Π'_D combining $a^{k(2n+1)}$ spikes from environment with a^i spikes, so that a rule with regular expression $a^{k(2n+1)+i}$ is applied, producing a^j spikes to Env. After applying such rule, a^j spikes remain in σ_1, and a single transition of D is simulated.

Note that the construction for Π'_D does not involve nondeterminism, and hence the previous technical problem: Let D have at least two transitions, (1) $\delta(q_i, b_k) = q_j$, and (2) $\delta(q_{i'}, b_{k'}) = q_{j'}$. We again let $j \neq j', i = k'$, and $k = i'$. Note that being a DFA, we have $i \neq k$. Observe that $k(2n+1)+i \neq k'(2n+1)+i'$. Therefore, Π'_D is deterministic, and has two rules r_1 and r_2 correctly

simulating (1) and (2), respectively. We now have the following result.

Theorem 6.2.3. *Any regular language L can be expressed as $L = L(\Pi'_D)$ for some 1-neuron SNP module Π'_D*

For a given DFST T, we construct an SNP module Π'_T simulating T as follows:

$$\Pi'_T = (\{a\}, \sigma_1, syn, \{1\}, \{1\}),$$

where

- $\sigma_1 = (1, \{a^{k(2n+1)+i+t}/a^{k(2n+1)+i+t-j} \rightarrow a^{n+s} | \delta'(q_i, b_k) = (q_j, c_s)\})$,
- $syn = \emptyset$.

We also have Π'_T as a 1-neuron SNP module similar to Π'_D. Again, $n, m, t \in \mathbb{N}$ are fixed numbers, and for each $1 \leq i \leq n, 1 \leq k \leq m$, and $1 \leq s \leq t$: each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented as $a^i, a^{k(2n+1)+t}$, and a^{n+s} spikes, respectively. The functioning of Π'_T is in parallel to Π'_D. Unlike Π_T, Π'_T is deterministic and correctly simulates T. We now have the next result.

Theorem 6.2.4. *Any finite transducer T can be simulated by some 1-neuron SNP module Π'_T.*

6.3 k-DFAO simulation and generating automatic sequences

Next, we modify the construction from Theorem 6.2.4 specifically for k-DFAOs by: (a) adding a second neuron σ_2 to handle the spikes from σ_1 until end of input is reached, and (b) using σ_2 to output a symbol once the end of input is reached. Also note that in k-DFAOs we have $t \leq n$, since each state must have exactly one output symbol associated with it. Observing k-DFAO and DFST definitions from Section 2.2, we find a subtle but interesting distinction as follows:

The output of the state after reading the last symbol in the input is the requirement from a k-DFAO, i.e. for every w over some Σ_k^8, the k-DFAO produces only one $c \in \Delta$ (recall the output function τ); In contrast, the output of DFSTs is a sequence of $Q \times \Delta$ (states and symbols), since $\delta(q_i, b_k) = (q_j, c_s)$. Therefore, if we use the construction in Theorem 6.2.4 for DFST in order to simulate k-DFAOs, we must ignore the first $|w| - 1$ symbols in the output of the system in order to obtain the single symbol we require.

For a given k-DFAO $M = (Q, \Sigma, \Delta, \delta, q_1, \tau)$, we have $1 \leq i, j \leq n$, $1 \leq s \leq t$, and $1 \leq k \leq m$. Construction of an SNP module Π_M simulating M, is as follows:

$$\Pi_M = (\{a\}, \sigma_1, \sigma_2, syn, \{1\}, \{2\}),$$

where

- $\sigma_1 = (1, R_1), \sigma_2 = (0, R_2)$,

- $R_1 = \{a^{k(2n+1)+i+t}/a^{k(2n+1)+i+t-j} \to a^{n+s} | \delta(q_i, b_k) = q_j, \tau(q_j) = c_s\}$
  $\cup \{a^{m(2n+1)+n+t+i} \to a^{m(2n+1)+n+t+i} | 1 \leq i \leq n\}$,
- $R_2 = \{a^{n+s} \to \lambda | \tau(q_i) = c_s\} \cup \{a^{m(2n+1)+n+t+i} \to a^{n+s} | \tau(q_i) = c_s\}$,
- $syn = \{(1, 2)\}$.

We have Π_M as a 2-neuron SNP module, and $n, m, t \in \mathbb{N}$ are fixed numbers. Each state $q_i \in Q$, each input symbol $b_k \in \Sigma$, and each output symbol $c_s \in \Delta$, is represented as a^i, $a^{k(2n+1)+t}$, and a^{n+s} spikes, respectively. In this case however, we add an end-of-input symbol \$ (represented as $a^{m(2n+1)+n+t}$ spikes) to the input string, i.e. if $w \in \Sigma^*$, the input for Π_M is $w\$$.

For any $b_k \in \Sigma$, σ_1 of Π_M functions in parallel to σ_1 of Π'_D and Π'_T, i.e. every transition $\delta(q_i, b_k) = q_j$ is correctly simulated by σ_1. The difference however lies during the step when \$ enters σ_1, indicating the end of the input. Suppose during this step σ_1 has a^i spikes, then those spikes are combined with the $a^{m(2n+1)+n+t}$ spikes from the environment. Then, one of the n rules in σ_1 with regular expression $a^{m(2n+1)+n+t+i}$ is applied, sending $a^{m(2n+1)+n+t+i}$ spikes to σ_2.

The first function of σ_2 is to erase, using forgetting rules, all a^{n+s} spikes it receives from σ_1. Once σ_2 receives $a^{m(2n+1)+n+t+i}$ spikes from σ_1, this means that the end of the input has been reached. The second function of σ_2 is to produce a^{n+s} spikes exactly once, by using one rule of the form $a^{m(2n+1)+n+t+i} \to a^{n+s}$. The output function $\tau(\delta(q_1, w\$))$ is therefore correctly simulated. We can then have the following result.

Theorem 6.3.1. *Any k-DFAO M can be simulated by some 2-neuron SNP module Π_M.*

Next, we establish the relationship of SNP modules and automatic sequences.

Theorem 6.3.2. *Let a sequence $\mathbf{a} = (a_n)_{n\geq 0}$ be k-automatic, then it can be generated by a 2-neuron SNP module Π.*

k-automatic sequences have several interesting robustness properties. One property is the capability to produce the same output sequence given that the input string is read in reverse, i.e. for some finite string $w = a_1 a_2 \dots a_n$, we have $w^R = a_n a_{n-1} \dots a_2 a_1$. It is known (e.g. [3]) that if $(a_n)_{n\geq 0}$ is a k-automatic sequence, then there exists a k-DFAO M such that $a_n = \tau(\delta(q_0, w^R))$ for all $n \geq 0$, and all $w \in \Sigma_k^*$, where $[w]_k = n$. Since the construction of Theorem 6.3.1 simulates both δ and τ, we can include robustness properties as the following result shows.

Theorem 6.3.3. *Let $\mathbf{a} = (a_n)_{n\geq 0}$ be a k-automatic sequence. Then, there is some 2-neuron SNP module Π where $\Pi(w^R\$) = a_n$, $w \in \Sigma_k^*$, $[w]_k = n$, and $n \geq 0$.*

An illustration of the construction for Theorem 6.3.1 is given in example 6.3.1.

Example 6.3.1. *(SNP module simulating the 2-DFAO generating the Thue-Morse sequence in Example 2.2.1) The SNP module is given in Figure 6.4, and we have $n = m = t = 2$.*

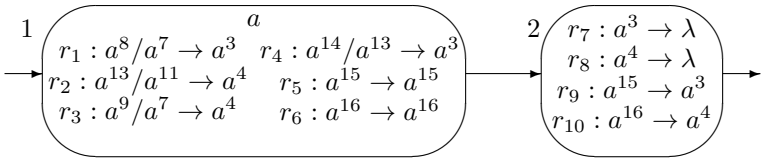

Figure 6.4: SNP module simulating the 2-DFAO in Figure 2.1.

Based on the construction for Theorem 6.3.1, we associate in Example 6.3.1 symbols 0 and 1 with a^7 and a^{12} spikes, respectively. The end-of-input symbol \$, q_1, and q_2 are associated with a^{14}, a, and a^2 spikes, respectively (with a and a^2 appearing only inside σ_1).

The 2-DFAO in Figure 2.1 has four transitions, and rules r_1 to r_4 simulate the four transitions. Rules r_5 and r_6 are only applied when \$ enters the system. Rules r_7 and r_8 are applied to "clean" the spikes from σ_1 while \$ is not yet encountered by the system. Rules r_8 and r_9 produce the correct output, simulating τ.

Chapter 7

SNP systems with structural plasticity (SNPSP systems)

In this chapter, the biological feature known as *structural plasticity* is first introduced in the framework of SNP systems. Structural plasticity refers to synapse creation and deletion, thus changing the synapse graph. The "programming" therefore of a brain-like model, the SNP system with structural plasticity (in short, SNPSP system), is based on how neurons connect to each other. SNPSP systems are also a partial answer to an open question on SNP systems with dynamism only for synapses. For both number accepting and generating modes, we prove that SNPSP systems are universal. Modifying SNPSP systems semantics, we introduce the spike saving mode and prove that universality is maintained. In saving mode however, a deadlock state can arise, and we prove that reaching such a state is undecidable.

7.1 Introduction

We are interested in this chapter with the biological feature known as *neural plasticity,* which is concerned with synapse modifications. Related to this feature are works in SNP systems with some forms of neural plasticity: Hebbian SNP (HSNP) systems [33] and SNP systems with neuron division and budding [69][95]. In HSNP systems, given two neurons σ_i and σ_j, and a synapse (i, j) between them, if spikes from neuron σ_i arrive repeatedly and shortly before neuron σ_j sends its own spikes, the synapse weight (strength) of (i, j) increases. If however the spikes from neuron σ_i arrive after the spikes of σ_j are sent, synapse weight of (i, j) decreases. HSNP systems were introduced for possible machine learning use in the framework of SNP systems.

In [69][95], other than spiking rules, two new rules are introduced: neuron division and neuron budding rules. Both new rules implicitly involve creation of novel synapses (*synaptogenesis*) as a result of the creation of novel neurons (*neurogenesis*). The initial synapse graph of the system is thus changed due to the application of the new rules, thus creating exponential workspace (in terms of neurons) in linear time. The `SAT` problem was then efficiently solved in [69][95] but by using the

workspace created.

The particular type of neural plasticity we introduce in the framework of SNP systems in this chapter is *structural plasticity.*[1] Structural plasticity is concerned with any change in connectivity between neurons, with two main mechanisms: (1) synaptogenesis and synapse deletion, (2) synaptic rewiring [5]. Unlike Hebbian (also known as *functional*) plasticity which concerns itself with only two neurons (for synapse strength modification), synaptic rewiring involves at least three neurons: if we have three neurons $\sigma_i, \sigma_j, \sigma_k$ and only one synapse (i, j), in order to create a synapse (i, k) then synapse (i, j) must be deleted first[2]. Synaptogenesis is present and implied during neuron division and budding due to neurogenesis, while synapse rewiring is not included, and synapse deletion is also implied.

The introduction of spiking neural P systems with structural plasticity (SNPSP systems for short) is also a response to the open problem **D** in [80] where "dynamism" only for synapses is to be considered. In summary, SNPSP systems are distinct from related variants due to the following: HSNP systems in [33] involve functional plasticity, instead of structural, so that there is no dynamism in their synapse graph; systems in [69][95] involve dynamism in both neurons and synapses (synapse creation and deletion are implicit by-products of neurogenesis), and no synaptic rewiring is involved. In contrast, SNPSP systems have a static collection of neurons, where the "channels" (the synapses) to workspaces (the neurons) is not static, due to structural plasticity.

Biological neurons in adult human brains can reach more than billions in numbers, and each neuron can wire to thousands of other neurons. This phenomenon is another biological motivation in this work. In this work we can have a collection of (possibly exponential number of) neurons, sometimes referred to as pre-computed resources as in [46]. The initial synapse graph can still be composed of a linear number of neurons wired or connected using synapses. However, we are only concerned with the creation and deletion of synapses over this collection for computing use. The synapse graph will then change: it is possible the system will connect an exponential number of neurons together at certain time steps (due to synapse creation) and make use of additional workspace (i.e. recently connected neurons); at other time steps the system can connect a linear or polynomial number of neurons (due to synapse deletion) and "release" additional workspace from the system which are no longer needed.

The standard SNP system originally from [44] included neurons with spiking rules that can have complex regular expressions, delays (in applying rules), and forgetting rules (rules that remove spikes from the system). A series of papers which proved computational universality (or simply universality, if there is no confusion) while simplifying the regular expressions of rules, removing delays or forgetting rules followed, e.g. [30][40], with the most recent being [66]. In [40] for example, SNP systems can be universal with the following being true: rules are without delays; without using

[1] First introduced in [12] and improved and extended in [13].

[2] This is inspired by *synaptic homeostasis* in biological neurons, where total synapse number in the system is left unchanged [5].

forgetting rules; regular expressions are simple. In [30] it was shown that universality is achieved with more restrictions: without delays and forgetting rules; without delays and simple regular expressions. In order to maintain universality while further simplifying the system, we look for other biological motivations.

In [70], universality in SNP systems was achieved with the following features: rules are without delays, and all neurons only have exactly one and the same simple rule. The way to control or "program" the SNP systems in [70] was using additional structures from neuroscience called astrocytes. Astrocytes introduce nondeterminism in the system, allowing the system to have simple neurons (that is, neurons in [70] contain only the rule $a \to a$).

In this chapter we use the biological feature of structural plasticity to achieve universality with the following *normal form* or simplifying restrictions: (a) only a "small" number of neurons have plasticity rules (details to follow shortly), (b) neurons without plasticity rules are simple (they only contain the rule $a \to a$), and (c) without the use of delays and forgetting rules. We do not include additional neuroscience structures other than neurons and their synapses. The introduction of the structural plasticity feature, in order to "program" the system, allows the system to maintain universality even with restrictions (a) to (c). Note that the idea of programming how a network of neurons connect in order to perform tasks is an old one in computer science, see for example [93].

Next, we modify the semantics of SNPSP systems, introducing a spike saving mode. For SNPSP systems operating in such a mode, we prove that their computing power is not diminished. However, an interesting property called a deadlock state, can occur in an SNPSP system in saving mode. We prove that reaching such a state in saving mode is undecidable for an arbitrary SNPSP system.

7.2 Spiking neural P systems with structural plasticity

In this section we introduce the variant of SNP systems with structural plasticity. The reader is invited to consult again with Chapter 4 or the original SNP systems paper in [44] for initial computing and bio-motivations and preliminary results.

Formally, a spiking neural P system with structural plasticity (SNPSP system) of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$, where:

- $O = \{a\}$ is the singleton alphabet (a is called spike)
- $\sigma_1, \ldots, \sigma_m$ are neurons of the form $(n_i, R_i), 1 \leq i \leq m$, with $n_i \geq 0$ indicating the initial number of spikes in σ_i, and R_i is a finite rule set of σ_i with the following forms:
  1. Spiking rule: $E/a^c \to a$, where E is a regular expression over O, with $c \geq 1$;
  2. Plasticity rule: $E/a^c \to \alpha k(i, N)$, where $c \geq 1$, $\alpha \in \{+, -, \pm, \mp\}$, $k \geq 1$, and $N \subseteq \{1, \ldots, m\}$
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, are synapses between neurons;

- $in, out \in \{1, \ldots, m\}$ indicate the input and output neurons, respectively.

Spiking rule semantics in SNPSP systems are similar with SNP systems in Chapter 4 and [44]. We do not use forgetting rules (rules of the form $a^s \to \lambda$) or rules with delays of the form $E/a^c \to a; d$ for some $d \geq 1$. Plasticity rules are applied as follows. If at step t we have that σ_i has $b \geq c$ spikes and $a^b \in L(E)$, a rule $E/a^c \to \alpha k(i, N) \in R_i$ can be applied. The set N is a collection of neurons to which σ_i can connect to or disconnect from using the applied plasticity rule. The rule consumes c spikes and performs one of the following, depending on α:

- If $\alpha = +$ and $N - pres(i) = \emptyset$, or if $\alpha = -$ and $pres(i) = \emptyset$, then there is *nothing more to do*, i.e. c spikes are consumed but no synapses are created or removed. Notice that with these semantics, a plasticity rule can replace at certain cases a forgetting rule, i.e. the former can be used to consume spikes without producing any spike.
- For $\alpha = +$, if $|N - pres(i)| \leq k$, *deterministically* create a synapse to every σ_l, $l \in N - pres(i)$. If however $|N - pres(i)| > k$, *nondeterministically* select k neurons in $N - pres(i)$, and create one synapse to each selected neuron.
- For $\alpha = -$, if $|pres(i)| \leq k$, deterministically delete all synapses in $pres(i)$. If however $|pres(i)| > k$, nondeterministically select k neurons in $pres(i)$, and delete each synapse to the selected neurons.

If $\alpha \in \{\pm, \mp\}$: create (respectively, delete) synapses at step t and then delete (respectively, create) synapses at step $t + 1$. Only the priority of application of synapse creation or deletion is changed, but the application is similar to $\alpha \in \{+, -\}$. Neuron i is always open from t until $t + 1$, but σ_i can only apply another rule at time $t + 2$.

An important note is that for σ_i applying a rule with $\alpha \in \{+, \pm, \mp\}$, creating a synapse always involves a sending of one spike when σ_i connects to a neuron. This single spike is sent at the time the synapse creation is applied, i.e. whenever synapse (i, j) is created between σ_i and σ_j during synapse creation, we have σ_i immediately transferring one spike to σ_j.

A system state or configuration of an SNPSP system is based on (a) distribution of spikes in neurons, and (b) neuron connections based on the synapse graph *syn*. We can represent (a) in the same way as SNP systems (with delays removed), i.e. as $\langle s_1, \ldots, s_m \rangle$ where $s_i, 1 \leq i \leq m$, is the number of spikes contained in σ_i. For (b) we can derive $pres(i)$ and $pos(i)$ from *syn*, for a given σ_i. The initial configuration therefore is represented as $\langle n_1, \ldots, n_m \rangle$, with the possibility of a disconnected graph, or even $syn = \emptyset$. A computation is again a sequence of configuration transitions from an initial configuration. A computation halts if the system reaches a halting configuration, where no rules can be applied. Whether a computation is halting or not, we associate natural numbers $1 \leq t_1 < t_2 < \ldots$ corresponding to the time instances when the neuron *out* sends a spike out to (or when *in* receives a spike from) the system.

A result of a computation which we consider in this chapter is as follows, as in [44]: we only consider the first two time instances t_1 and t_2 that σ_{out} spikes. Their difference, i.e. the number

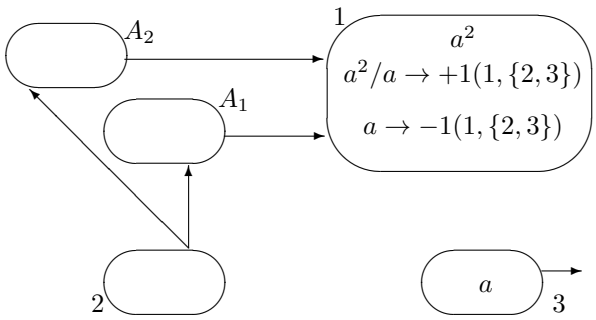

Figure 7.1: An SNPSP system Π_{ex}.

$t_2 - t_1$, is said to be computed by Π. We denote the set of all numbers computed in this manner by Π as $N_2(\Pi)$.

In $N_2(\Pi)$ the neuron *in* is ignored, and we refer to this as the generative mode. SNPSP systems in the accepting mode ignore σ_{out} and work as follows: The system begins with an initial configuration, and exactly two spikes are introduced in the system (using σ_{in}) at times t_1 and t_2. The number $t_2 - t_1$ is accepted by Π if the computation halts. The set of numbers accepted by Π is denoted as $N_{acc}(\Pi)$. The families of all sets of $N_\alpha(\Pi)$, with $\alpha \in \{2, acc\}$, is denoted as $N_\alpha SNPSP$.

7.3 An example of an SNPSP system

In this section, we provide an example to illustrate the definition and semantics of an SNPSP system. Consider an SNPSP system Π_{ex} shown in Figure 7.1. Neurons $2, out = 3, A_1$, and A_2 contain only the rule $a \to a$ and we omit this from writing. In the initial configuration, at some time t, is where only σ_1 has two spikes and σ_3 has only one spike. Neuron 1 is the only neuron with a plasticity rule, where $\alpha \in \{+, -\}$, and we have $syn = \{(2, A_1), (2, A_2), (A_1, 1), (A_2, 1)\}$.

Π_{ex} operates as follows: The application of the two rules in σ_1 are deterministic. Nondeterminism in Π_{ex} exists only in selecting which synapses to create to or delete from.[3] Neuron 3 has $n_3 = 1$, so it sends a spike out to the environment, and we label this event as time t_1. Since $n_1 = 2$, the rule with $\alpha = +$ is applied, so σ_1 nondeterministically selects one (since $k = 1$) of σ_2 or σ_3 to create a synapse to.

If the synapse $(1, 3)$ is created, then σ_1 sends one spike to σ_3 also at time t_1. The rule with $\alpha = +$ consumes one spike, so σ_1 now has one spike. At time t_2 the rule with $\alpha = -$ is applied, and synapse $(1, 3)$ is deleted. The spike that σ_3 received from σ_1 is sent to the environment at time t_2 also. Two spikes are sent out to the environment by Π_{ex} before it halts. The computed number in this case is then $t_2 - t_1 = 1$. The computation of $\{1\}$ by Π_{ex} is shown in table 7.1, where (!) means that the output neuron spikes to the environment.

[3] We refer to this later as *synapse level nondeterminism*

time	σ_1	σ_2	σ_3	σ_{A_1}	σ_{A_2}	syn
t_0	2	0	1	0	0	syn
t_1	1	0	1(!)	0	0	$syn' = syn \cup \{(1,3)\}$
t_2	0	0	0(!)	0	0	$syn = syn' - \{(1,3)\}$

Table 7.1: Computation of Π_{ex} computing $\{1\}$.

time	σ_1	σ_2	σ_3	σ_{A_1}	σ_{A_2}	syn
t_0	2	0	1	0	0	syn
t_1	1	1	0(!)	0	0	$syn'' = syn \cup \{(1,2)\}$
t_2	0	0	0	1	1	$syn = syn'' - \{(1,2)\}$
t_3	2	0	0	0	0	syn
t_4	1	0	1	0	0	$syn' = syn \cup \{(1,3)\}$
t_5	0	0	0(!)	0	0	$syn = syn' - \{(1,3)\}$

Table 7.2: Computation of Π_{ex} computing $\{4\}$.

If however the synapse $(1,2)$ is created, σ_1 sends a spike to σ_2 at time t_1 during synapse creation. Neuron 2 then sends one spike each to auxiliary neurons A_1 and A_2 at time t_2. Also at time t_2 is when the rule with $\alpha = -$ is applied, and $(1,2)$ is deleted. A_1 and A_2 send one spike each to σ_1, so that σ_1 has two spikes again at time t_3, as in the initial configuration. As long as σ_1 creates synapse $(1,2)$ instead of $(1,3)$ then Π_{ex} keeps receiving two spikes in a loop.

Notice that if at some time $m > 1$ the synapse $(1,2)$ is created, it will take σ_1 three time steps for the possibility of creating $(1,3)$ again. Once $(1,3)$ is created, one more step is required for σ_3 to spike to the environment for the second and final time. The computation of $\{4\}$ for example, is shown in table 7.2. Therefore, from the operation of Π_{ex} we have $N_2(\Pi_{ex}) = \{1,4,7,10,\ldots\} = \{3m+1 | m \geq 0\}$. Also notice that for Π_{ex}, its two plasticity rules in σ_1 can be replaced by a single plasticity rule: $a^2 \to \pm 1(1,\{2,3\})$.

7.4 Universality of SNPSP Systems

In this section we show that SNPSP systems are computationally universal, i.e. they characterize NRE, both in the accepting and generative modes. In SNPSP systems there are two types of nondeterminism: (1) in selecting which rule to apply in a neuron (rule level nondeterminism), and (2) in selecting which synapses to create to or delete from (synapse level nondeterminism). However, we prove that synapse level nondeterminism is sufficient for universality. Therefore, the SNPSP systems we construct in this work do not involve rule level nondeterminism.

We start with SNPSP systems working in the generative mode, followed by the accepting mode. As in Π_{ex} and unless mentioned otherwise, we omit from writing the rules for simple neurons with $a \to a$ as the only rule. A network of these simple neurons (first referred to in [70]) with rules of a simple form, together with fewer neurons that have plasticity rules, are computationally powerful:

they can perform tasks that Turing machines can perform.

SNPSP systems working in the generative mode

Theorem 7.4.1. $NRE = N_2SNPSP$.

Proof. To prove Theorem 7.4.1 we only need to prove $NRE \subseteq N_2SNPSP$ by simulating a register machine M in generative mode with an SNPSP system. The converse, i.e. $N_2SNPSP \subseteq NRE$, is straightforward or the Turing-Church thesis can be invoked. Without loss of generality we may assume for register machine $M = (m, I, l_0, l_h, R)$ to have: (a) all registers except register 1 are empty at halting; (b) the output register is never decremented during any computation; and (c) the initial instruction of M is an `ADD` instruction.

The SNPSP system Π simulating M has three modules: the ADD, SUB and FIN modules shown in Figures 7.2, 7.3, and 7.4 respectively. All three modules consist of simple neurons and some neurons that have plasticity rules. The ADD and SUB modules simulate the `ADD` and `SUB` instructions of M, respectively. The FIN module is used to output the computation of Π in the generative mode mentioned in section 7.2.

Given a register r of M we have an associated neuron σ_r in Π. In any computation, if r stores the value n, then σ_r will contain $2n$ spikes. For every label l_i in M we have σ_{l_i} in Π. The initial configuration of Π is where only σ_{l_0} has one spike, indicating that instruction l_0 in M is to be executed. Simulating a rule $l_i : (\texttt{OP}(r) : l_j, l_k)$ in M means σ_{l_i} has one spike and is activated. Then, depending on the operation $\texttt{OP} \in \{\texttt{ADD}, \texttt{SUB}\}$, an operation is performed on σ_r. Either σ_{l_j} or σ_{l_k} receives a spike and becomes activated. When M executes l_h (the label for the halting instruction of M), Π completely simulates the computation of M. Π then activates σ_{l_h}, and σ_{out} sends two spikes to the environment with a time difference equal to the stored value in register 1 (the output register).

Module ADD shown in Figure 7.2 simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$: The initial instruction of M is labeled l_0, and it is an `ADD` instruction. Assume that instruction $l_i : (\texttt{ADD}(r) : l_j, l_k)$ is to be simulated at time t. Therefore σ_{l_i} contains one spike while all other neurons are empty except neurons associated with registers. Neuron l_i uses its rule $a \to a$ and sends one spike each to neurons l_i^1 and l_i^2. At time $t + 1$ both neurons l_i^1 and l_i^2 send one spike to σ_r so that σ_r receives two spikes corresponding to the value one in register r of M. Also at time $t + 1$ is when neuron l_i^2 sends one spike to σ_p, which is the only neuron in module ADD with plasticity rules.

Once σ_p receives a spike, σ_p consumes one spike and at time $t+2$, σ_p nondeterministically creates one synapse (since $\alpha = \pm, k = 1$) to either neuron l_j or l_k. Also at time $t + 2$, σ_p sends a spike to either neuron l_j or l_k; If synapse (p, l_j) was created then σ_p removes (p, l_j) afterwards, otherwise σ_p removes (p, l_k), at time $t + 3$.

At time $t+3$ either neuron l_j or l_k is activated, i.e. receives one spike from σ_p, and can therefore perform the associated instruction. From firing neuron l_i the module ADD increments the spikes

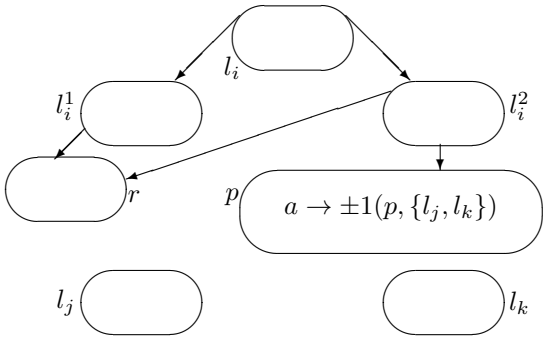

Figure 7.2: Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$.

contained in σ_r by two and nondeterministically activates either neuron l_j or l_k. We therefore correctly simulate an ADD instruction.

Module SUB shown in Figure 7.3 simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$: We simulate a SUB instruction with the SUB module as follows: Initially only σ_{l_i} has one spike and all other neurons are empty except those associated with registers. Let t be the time when neuron l_i fires, using its rule $a \to a$ to send one spike to σ_r and $\sigma_{l_i^1}$. As with the SUB instruction in M, the two cases are when register r stores a nonzero value (execute instruction l_j) or when r stores zero (execute l_k). For the moment let us set $N_j = \{l_i^2\}$ and $N_k = \{l_i^3\}$ so that $|N_j| = |N_k| = 1$. The case for a general N_j and N_k will be explained shortly.

If σ_r contains a nonzero number of spikes at time t, then it contains $2n$ (even numbered) spikes corresponding to n stored in register r. At time $t+1$ neuron r has $2n+1$ (odd numbered) spikes, the additional one spike will come from neuron l_i at time t. The rule with regular expression $a(a^2)^+$ is enabled at $t+1$ since σ_r has an odd number of spikes. At time $t+1$ the rule consumes three spikes and deterministically creates $|N_j| = 1$ synapse (r, l_i^2), and sends one spike to $\sigma_{l_i^2}$. At time $t+2$, synapse (r, l_i^2) is deleted (since $\alpha = \pm$). At time $t+2$ therefore, neuron l_i^2 contains two spikes: one each from neurons l_i^1 and r. We have neuron l_i^2 creating at time $t+2$ a synapse, as well as sending one spike, to neuron l_j. Neuron l_j is therefore activated.

Removing three spikes from $2n+1$ spikes in σ_r makes the number of spikes contained in σ_r even again. In particular, after rule application, σ_r will contain $2(n-1)$ spikes corresponding to $n-1$ in r. Module SUB correctly simulates decreasing the value stored in r by one if r stored a nonzero value, and then executing l_j.

If σ_r contains no spikes at time t corresponding to a stored value of zero in r, then σ_r contains one spike at time $t+1$ and applies the rule $a \to \pm 1(r, \{l_i^3\})$. This rule consumes one spike, reducing the spikes contained in σ_r back to zero. The rule also creates synapse (r, l_i^3) and sends one spike to neuron l_i^3 at time $t+1$. At $t+2$, synapse (r, l_i^3) is deleted. Neuron l_i^3 contains 2 spikes at $t+1$, one each from neurons r and l_i^1. At $t+2$, neuron l_i^3 creates synapse (l_i^3, l_k) and sends one spike to σ_{l_k}, activating σ_{l_k}. Module SUB correctly simulated maintaining the zero stored in r if r initially stored

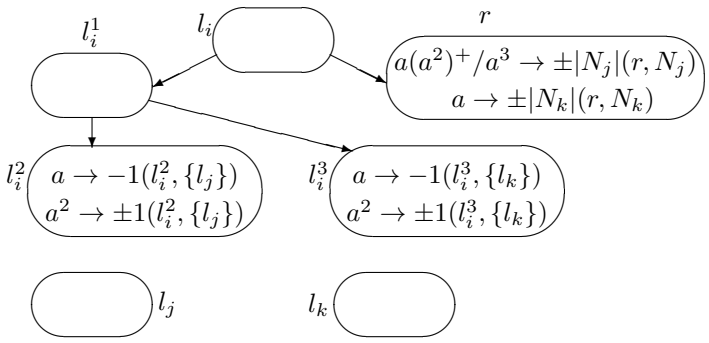

Figure 7.3: Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$.

zero, and then executing l_k.

For the general case where there are at least two SUB instructions operating on register r, this means we have at least two SUB modules operating on the same σ_r: to simulate $l_x : (\texttt{SUB}(r) : l_{x_1}, l_{x_2})$ and $l_y : (\texttt{SUB}(r) : l_{y_1}, l_{y_2})$, we have to be certain that simulating l_x (l_y, respectively) only activates either neuron l_{x_1} or l_{x_2} (l_{y_1} or l_{y_2} respectively). We define N_k (N_j, respectively) as $N_k = \{l_i^2 | l_i$ is a SUB instruction on $r\}$ ($N_j = \{l_i^3 | l_i$ is a SUB instruction on $r\}$, respectively).

The possibility of at least two SUB modules interfering with each other is removed since each l_i^2 and l_i^3, $i \in \{x, y\}$, is only activated when a spike from the corresponding l_i^1 is received. Only the specific SUB instruction to be simulated has its corresponding neuron l_i^2 (otherwise l_i^3, depending on the stored value in r) in a SUB module receive two spikes. The other SUB modules only have their neuron l_i^2 (otherwise, l_i^3) receive only one spike. Either neuron l_i^2 or l_i^3 therefore activate the correct neuron: whenever $\alpha = \pm$ (when containing two spikes), or $\alpha = -$ (when containing one spike). If $\alpha = -$ then only synapse deletion is performed. Therefore, the SUB instruction is correctly simulated by the SUB module.

Module FIN shown in Figure 7.4 once l_h is reached in M: We assume at time t that the computation in M halts, so that instruction with label l_h is reached. Also at time t we have σ_1 containing $2n$ spikes corresponding to the value n stored in register 1 of M. Neuron l_h sends one spike each to σ_1 and σ_{out}. At time $t+1$ neuron *out* sends the first of two spikes that it will send to the environment before computation halts. Also at time $t+1$ we have σ_1 containing $2n+1$ spikes. Neuron 1 continuously applies its rule $a^3(a^2)^+/a^2 \to +1(1, \{out\})$ if σ_1 initially contained four or more spikes. The rule performs the following every time it is applied: two spikes are consumed and the synapse $(1, out)$ is deleted (since $\alpha = -, k = 1$).

Notice that synapse $(1, out)$ does not exist, so the two spikes consumed are simply removed from the system and no synapse is actually deleted. Also notice that applying this rule leaves σ_1 with an odd number of spikes afterwards. The value of k in the rule can actually be any positive integer but in this case it is simply set to one. Once the number of spikes in σ_1 is reduced to three, the rule $a^3 \to \pm1(1, \{out\})$ is applied. This rule is applied after n applications of the previous rule. At time

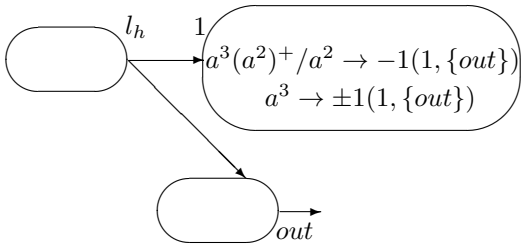

Figure 7.4: Module FIN.

$t+n$ the rule removes three spikes and leaves no spikes in σ_1, creates a synapse and sends a spike to σ_{out}, and deletes the synapse at $t+n+1$. Neuron *out* receives one spike from σ_1 and spikes for the second and final time to the environment at time $t+n+1$. The time difference between the first and second spiking of σ_{out} is $(t+n+1)-(t+1)=n$, exactly the number stored in register 1 of M when the computation of M halted.

From the description of the operations of modules ADD, SUB, and FIN, Π clearly simulates the computation of M. Therefore we have $N_2(\Pi) = N(M)$ and this completes the proof. □

SNPSP systems working in the accepting mode

SNPSP systems in the accepting mode ignore the *out* neuron and use an *in* neuron to take in exactly two spikes. The time difference between the two input spikes is the number computed by the system, if the computation halts. Recall that a register machine M is computationally universal even for the deterministic accepting case. The resulting SNPSP system simulating M is therefore simpler (i.e. contains smaller number of parameters e.g. number of rules) compared to the system simulating the generative case as in Theorem 7.4.1.

Theorem 7.4.2. *$NRE = N_{acc}SNPSP$.*

Proof. The proof for Theorem 7.4.2 is a consequence of the proof of Theorem 7.4.1. Given a deterministic register machine $M = (m, I, l_0, l_h, R)$ we construct an SNPSP system Π as in the proof of Theorem 7.4.1. Since M is deterministic in this case, we modify Π so that addition is deterministic and we use an INPUT module instead of a FIN module. As with the proof of Theorem 7.4.1, the modules will contain simple neurons having only $a \to a$ as their rule, and some neurons with plasticity rules.

The new module INPUT is shown in Figure 7.5 and functions as follows: All neurons are initially empty and we let the time that the first spike enters the module be t. The second and final spike input will arrive at time $t+n$ so that $(t+n)-t=n$ is the value stored in register 1 of M. Neuron *in* uses its rule $a \to a$ at time $t+1$ to send one spike each to neurons A_1, A_2, and A_3.

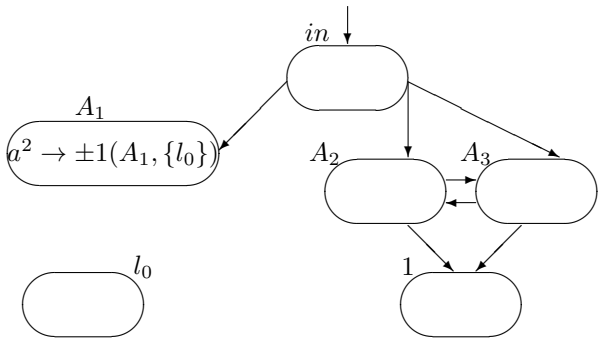

Figure 7.5: Module INPUT.

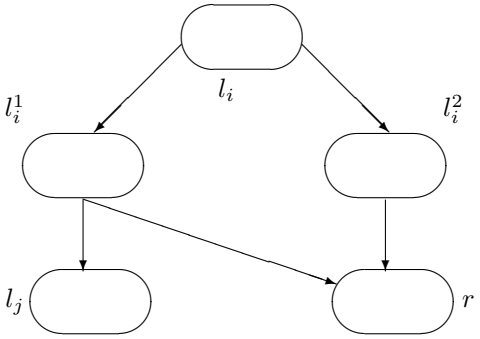

Figure 7.6: Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j)$.

Neurons A_2 and A_3 exchange spikes every step from $t+2$ until $t+n+1$. During every exchange, neurons A_2 and A_3 increase the spikes inside σ_1 by two. Once the second and the final spikes arrive, σ_{in} spikes at $t+n+1$ so that A_2 and A_3 now each have two spikes, and therefore cannot apply their only rule $a \to a$. At time $t+n+1$ neuron A_1 collects two spikes so it can apply its plasticity rule to activate neuron l_0. The activation of neuron l_0 corresponds to the start of the simulation of instruction l_0 in M. Note that from time $t+2$ until $t+n+1$, σ_1 collects a total of $2n$ spikes, corresponding to the value n stored in register 1.

Since for the accepting case M can be computationally universal even with a deterministic ADD instruction, we have the deterministic ADD module in figure 7.6. The functioning of the deterministic ADD module is clear and is simpler than the nondeterministic ADD module. The SUB module remains the same as in the proof of Theorem 7.4.1. Module FIN is not used, while σ_{l_h} remains in the system except that $pres(l_h) = \emptyset$.

Once σ_{l_h} receives a spike indicating that the computation of M has halted, the neuron applies its rule $a \to a$ but does not send its spike to any neuron. Therefore, the computation in Π halts only if the computation in M halts, and we have $N_{acc}(\Pi) = N(M)$ completing the proof. □

7.5 Spike saving mode universality and deadlock

In this section we consider a modification of how plasticity rules are applied: if there is nothing more to do in terms of synapse creation or deletion, we do not consume a spike. We prove that given this modification, SNPSP systems are still universal. Additionally, an interesting property, the deadlock, can occur.

Let us consider again the definition and semantics of plasticity rules in Section 7.2: If σ_i contains b spikes and a rule $E/a^c \to \alpha k : (i, N_j)$, if $a^b \in L(E)$, then the plasticity rule can be applied. Applying the rule means consuming c spikes, and then performing plasticity operations depending on the value of α and k. If however $pres(i) = \emptyset$, there is nothing more to do for $\alpha = -$: This is because there is no synapse to delete, since σ_i is not a presynaptic neuron of any other neuron. Such a case is seen with the module FIN in Figure 7.4. Note that c spikes are still consumed, even if we later realize that there is nothing more to do. A similar case also exists for $\alpha = +$.

An interesting modification of this plasticity rule mode of operation is as follows: during the "nothing more to do" case, i.e. whenever $\alpha \in \{+, -\}$ and $N_j - pres(i) = \emptyset$ or $pres(i) = \emptyset$, we *save spikes* so that they are not consumed. As an analogy, if we pay a certain money or currency[4] to perform a task, then this saving mode means our money is returned if we discover that the task has been performed already. In contrast, the nonsaving mode in Section 7.2 is analogous to paying money to perform a task, regardless if the task is actually performed or not.

A natural inquiry from modifying SNPSP systems from nonsaving to saving mode is: are SNPSP systems in the saving mode still universal? The answer is affirmative, and requires a modification of the FIN module in Figure 7.4. The modification is the basis for the next result. We denote by $SNPSP_s$ the family of sets computed by an SNPSP system in saving mode.

Theorem 7.5.1. $NRE = N_2SNPSP_s$

Proof. The SUB and FIN modules make use of the nonsaving mode in Theorem 7.4.1. We only need to modify the two modules, while the ADD module remains intact. We denote the modified SUB and FIN modules as SUB′ and FIN′, respectively. Module FIN′ is shown in Figure 7.7. For FIN′, we add one new neuron σ_p and a new synapse $(1, p)$. Neuron p does not have any rules and $pres(p) = \emptyset$. We also remove the plasticity rule $a^3(a^2)^+/a^2 \to -1(1, \{out\})$ and replace it with a spiking rule $a^3(a^2)^+/a^2 \to a$.

Note that we still do not use forgetting rules. The purpose of the new rule in σ_1 is to reduce the number of spikes each time step by two in σ_1 until only three remain. Once there are three spikes in σ_1, then the remaining rule $a^3 \to \pm 1(1, \{out\})$ is applied. The new σ_p functions as a trap or repository for the spikes removed from σ_1. We have FIN′ functioning as it should be: delaying the second spike of σ_{out} for n steps, as required, and as was performed by FIN also.

[4] Or in the case of neurons, the quantum of energy which is the spike.

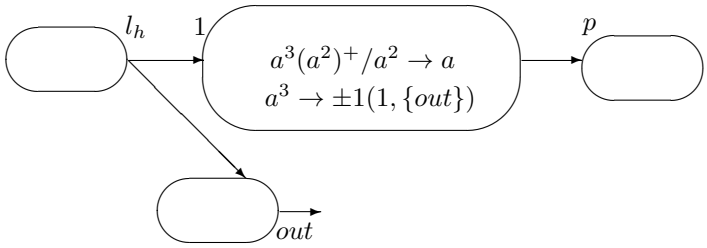

Figure 7.7: Module FIN′.

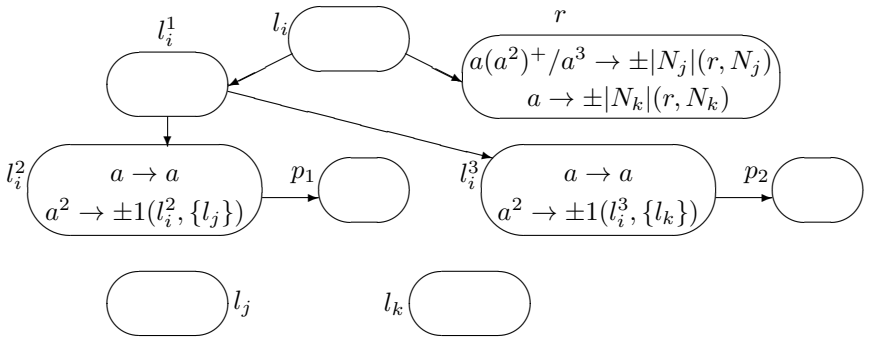

Figure 7.8: Module SUB′ simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$.

Module SUB′ is shown in Figure 7.8. For SUB′, we add trap neurons σ_{p_1} and σ_{p_2} for every neuron l_i^2 and l_i^3 in a SUB module, respectively. We also add for each SUB module, the synapses (l_i^2, p_1) and (l_i^3, p_2). We replace the rule $a \to -1(l_i^2, \{l_j\})$ in neuron l_i^2 (rule $a \to -1(l_i^3, \{l_k\})$ in neuron l_i^3, respectively) with a spiking rule $a \to a$. Similar to the trap neuron in FIN′, the trap neurons in SUB′ receive the unwanted spikes from neuron l_i^2 (or l_i^3) so that interference does not occur. Therefore, SUB′ also correctly simulates a SUB instruction as required. □

Corollary 7.5.1. $NRE = N_{acc}SNPSP_s$

Corollary 7.5.1 follows from Theorem 7.5.1, since in the accepting mode from Theorem 7.4.2, the saving or nonsaving mode is of no consequence. An interesting property that arises when using the saving mode is the existence of a *deadlock*. During the saving mode, deadlock occurs when a plasticity rule is applied because the regular expression is satisfied, but no actual work is performed by the neuron. The result is a state where no further computation can continue, and the system is stuck or trapped in the current configuration, even if a rule can always be applied. More formally, a deadlock state (or configuration) occurs if a σ_i can apply at least one plasticity rule and we have the following sequence of events during the same time step: a^c spikes are consumed, however the applied plasticity rule either has $\alpha = +$ and $N - pres(i) = \emptyset$, or $\alpha = -$ and $pres(i) = \emptyset$; since there

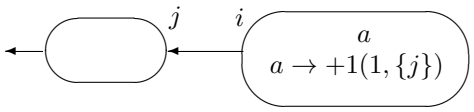

Figure 7.9: The SNPSP system $\Pi_{(+)}$ with a deadlock from the proof of lemma 7.5.1.

is nothing more to do in such a case, a^c spikes are returned to σ_i.

Lemma 7.5.1. *A deadlock can exist in an arbitrary SNPSP system in saving mode having at least two neurons, with* $\alpha \in \{+, -\}$.

Proof. Let us first consider when $\alpha = +$, and we define a $\Pi_{(+)}$ as follows.

$$\Pi_{(+)} = (\{a\}, \sigma_i, \sigma_j, syn_{(+)})$$

where $\sigma_i = (1, R_i)$, $R_i = \{a \to +1(i, \{j\})\}$, $\sigma_j = (0, \emptyset)$, and $syn_{(+)} = \{(i, j)\}$. Figure 7.9 provides an illustration for $\Pi_{(+)}$. We have that $\Pi_{(+)}$ has exactly one synapse between σ_i and σ_j, and exactly one rule (a plasticity rule) in σ_i. Once the plasticity rule is applied, the saving mode requires us to consume one spike first. The plasticity rule has $\alpha = +$ and $k = 1 = |pres(i)|$. Therefore we must create a new synapse from σ_i to σ_j. However, the synapse (i, j) already exists and we return the consumed spike, as dictated by the saving mode. Therefore $\Pi_{(+)}$ is stuck in the current and only configuration: the regular expression of the only rule is satisfied, however, we cannot perform further computations.

Now let us consider a similar SNPSP system $\Pi_{(-)}$ defined as follows.

$$\Pi_{(-)} = (\{a\}, \sigma_i, \sigma_j, syn_{(-)})$$

where $\sigma_i = (1, R_i)$, $R_i = \{a \to -1(i, \{j\})\}$, $\sigma_j = (0, \emptyset)$, and $syn_{(-)} = \emptyset$. In a similar way, the one and only rule of $\Pi_{(-)}$ is always applied, since the regular expression is satisfied. However, there is no synapse between σ_i and σ_j, since $syn_{(-)} = pres(i) = \emptyset$, so there is nothing to do. The saving mode dictates that the consumed spike must be returned and the system remains in a deadlock. □

An important problem is whether it is decidable (i.e. a Turing machine halts and produces an output given an input) to have an arbitrary SNPSP system in saving mode arrives at a state of deadlock or not. Not surprisingly,[5] the answer to this problem is a negative one, as given by the next result.

Theorem 7.5.2. *It is undecidable whether an arbitrary SNPSP system* Π' *in saving mode, with at least two neurons, reaches a deadlock.*

[5]See e.g. Rice's Theorem in [37].

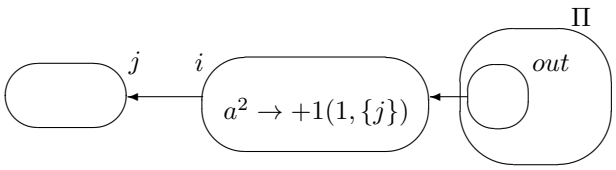

Figure 7.10: The SNPSP system Π' for the proof in Theorem 7.5.2.

Proof. We consider an arbitrary recursively enumerable set of natural numbers which we denote D. We can have a register machine, an SNP system, or an SNPSP system generate the elements of D. For simplicity, we choose to construct an SNPSP system Π similar to section 7.4 such that $D = N_2(\Pi)$. The system Π produces two spikes (using its output neuron) if and only if D is non-empty, and this task is undecidable. Furthermore, we construct an SNPSP system Π' using Π as a submodule. We refer to Figure 7.10 for a graphical representation for Π' instead of a formal definition.

If D is nonempty (this means σ_{out} of Π spikes twice), then σ_i receives two spikes and it can apply its rule. However, from lemma 7.5.1 we know that the application of the rule in σ_i results in a state of deadlock. The problem of realizing if D is nonempty is undecidable. Therefore, a deadlock is reached if and only if the set D is nonempty, which happens to be undecidable. □

Chapter 8

Sequential SNPSP systems induced by max/min spike number

SNPSP systems represent a class of SNP systems that have dynamism only for the synapses, i.e. neurons can use plasticity rules to create or remove synapses. In this chapter, we impose the restriction of sequentiality on SNPSP systems, using four modes: max, min, max-pseudo, and min-pseudo sequentiality. We also impose a normal form for SNPSP systems as number acceptors and generators. Conditions for (non)universality are then provided. Specifically, acceptors are universal in all modes, while generators need a nondeterminism source in two modes, which in this chapter is provided by the plasticity rules.

8.1 Introduction

Some of the common biologically inspired restrictions imposed on SNP systems and its variants are the notions of *simplicity* and *homogeneity*, which we recall here: a neuron is simple if it precisely has one rule; an SNP system is simple if all neurons are simple; an SNP system is homogeneous if all of its neurons precisely have the same set of rules; Simple SNP systems usually require more neurons to achieve universality due to the simplicity of their neurons [90][103]. Homogeneous SNP systems can become more "compact", i.e. they can require less neurons, due to the fact that they can have more than one rule in a neuron as in [48] and [100]. Both notions can have interesting biological and computational interpretations, for theoretical and practical use: the structure, i.e. the connectivity, of the neurons in the system is important for achieving a certain level of computing power; also, neurons do not need to be very complex (in the case of simple systems) or do not need to be very varied (in the case of homogeneous systems).

Another restriction imposed on SNP systems and its variants is the type(s) of rule(s) present in a neuron: standard (spiking) rules, extended (spiking) rules, spiking rules with delays, or forgetting rules. Extended rules allow for more compact systems in terms of neuron count, due to the ability to produce more than one spike each step as in [21] and [78]. Both forgetting rules and rules with

delays were used in [44], while in [66] for example, it was shown that universality is still achieved given the normal form of having no forgetting rules and rules with delays.

In this chapter we continue investigating the computing power of SNPSP systems. SNPSP systems represent the class of SNP systems that explicitly focus on synapse graph dynamism only, since the collection of neurons in the system remains static. Other than standard rules, the only other type of rule in SNPSP systems are plasticity rules: rules that allow neurons to create or delete synapses.

The restriction we impose on SNPSP systems in this chapter is sequentiality. Specifically and as in [41], we *induce* sequential operation based on the number of spikes stored in a neuron. For example, in the max sequentiality or $maxs$ mode, if neurons σ_a, σ_b, and σ_c contain 1, 3, and 2 spikes at the same step respectively, only σ_b is allowed to apply its rule. If however σ_c also stored 3 spikes, only one among σ_b or σ_c will apply its rule (nondeterministically chosen). Our results also exhibit (biologically and computationally) two interesting and simplifying features, also known as a normal form, as in [66] and [90]: a neuron has two rules (the maximum per neuron) if and only if it is purely plastic, i.e. only contains plasticity rules; if a neuron has a standard rule, then it is simple. These two features are interesting because while certain biological neurons seem to have more specific or complex functions (e.g. they create or remove synapses), other neurons seem more simple or generic (e.g. they simply function as spike repositories or relays). More related normal forms in this chapter include: almost simple SNP or ASSNP systems in [90], where the system has only one neuron that is not simple; simple and homogeneous SNP systems with astrocytes or SHSNPA systems in [70], where SHSNPA systems maintained universality despite being simple and homogeneous, due to the use of additional neuroscience structures (the astrocytes).

Another distinction of SNPSP systems, again biologically and computationally motivated, is an alternative source for nondeterminism. A more common source of nondeterminism in SNP systems is rule-level (in short, nd_{rule}): if more than one rule can be applied in a step, only one is chosen nondeterministically. In SNPSP systems, there is instead synapse-level nondeterminism (in short, nd_{syn}): selecting which synapses a neuron will create or remove. Our results show the conditions, with or without nd_{syn} (among other levels, except nd_{rule}) by which our systems become (non)universal. Our systems, under the normal form mentioned above, are investigated under four modes: max, min, max-pseudo, and min-pseudo sequentiality.

8.2 Sequential SNPSP systems based on spike number

In this section we introduce the idea of sequential operation on SNPSP systems. Let t be some step during a computation: we say a σ_i is *activated* at step t if there is at least one $r \in R_i$ that can be applied (i.e. the stored spikes in σ_i satisfy the regular expression of r); we say a σ_i is *active* at t if for some step $t' < t$, σ_i was activated by applying $r \in R_i$, and is currently fulfilling the requirements

for the application of r. Also, we say σ_i is *simple* if $|R_i| = 1$. Given an SNPSP system Π, we can have the following levels of nondeterminism: *system-level*, if given at least two activated neurons, Π allows exactly one neuron to fire; *rule-level*, if at least one neuron has at least two rules with regular expressions E_1 and E_2 such that $E_1 \neq E_2$ and $L(E_1) \cap L(E_2) \neq \emptyset$; *synapse-level*, if at least one neuron can nondeterministically create or delete a synapse.

By default SNP and SNPSP systems are locally sequential (at most one rule is applied per neuron) but globally parallel (all activated neurons must apply a rule). Note that the application of rules in neurons are synchronized, i.e. a global clock is assumed and if a neuron can apply a rule then it must do so.

A result of a computation can be defined in several ways in SNP systems literature. For sequential SNPSP systems, we use the following as in [48]: we only consider the first two consecutive steps t_1 and t_2 that σ_{out} spikes. Their difference minus 1, i.e. the number $n = (t_2 - t_1) - 1$, is said to be computed by Π. This way of encoding the output is a minor modification from the previous chapter where 1 is not subtracted from the time difference. We refer to Π as generator, if Π computes in this manner. Π can also work as an acceptor, as follows: n spikes are stored in a defined neuron in an initial configuration. Π then accepts n if the computation halts. This way of initializing the computation of an accepting system is also a minor modification from the previous chapter, where we remove the need for an input module as in [41].

The following two features are used in our systems as the normal form: (i) only purely plastic neurons (i.e. neurons with plasticity rules only) have at most two rules (the maximum in any neuron of the system), and (ii) neurons with standard rules are simple. We denote the family of sets computed by nondeterministic SNPSP systems (in the mentioned normal form) as generators as $N_{2,gen}SNPSP^{\gamma}$: subscript 2 indicates the first 2 spikes of σ_{out} as the result; we replace 2 and gen with acc for acceptors; we have the mode $\gamma \in \{maxs, maxps, mins, minps\}$ (details given shortly); Given an SNPSP system Π, we denote the computed set by Π as $N_m(\Pi)$, $m \in \{gen, acc\}$. To further parametrize our results in this chapter we have the following: $+syn_k$ ($-syn_j$, respectively) where at most k (j, resp.) synapses are created (deleted, resp.) each step; $nd_\beta, \beta \in \{syn, rule, sys\}$ indicate additional levels of nondeterminism; $rule_m$ indicates at most m rules (either standard or plasticity) per neuron; Since our results for k and j for $+syn_k$ and $-syn_j$ are equal, we write them instead in the compressed form $\pm syn_k$, where $\pm$ in this sense is not the same as when $\alpha = \pm$.

Induced sequentiality is as follows: In one step of an SNPSP system Π, only the neuron with the maximum (minimum, resp.) number of stored spikes becomes activated in $maxs$ ($mins$, resp) mode. If there is more than one such neuron, then exactly one is chosen among them (i.e. nd_{sys} is used). In *max pseudosequential* or $maxps$ (*min pseudosequential* or $minps$, resp.) mode, Π allows all neurons with the maximum (minimum, resp.) stored spikes to become activated (i.e. no nd_{sys}). In all four modes, the system is sequential, induced by the global maximum (minimum, resp.) spike number. The nd_{sys} was used in [41] (denoted as nondeterminism at the level of the system) and [47].

In [41], they denoted nd_{rule} as nondeterminism at the level of neurons. Note that if γ for example takes the value $maxs$, nd_{sys} is implied so that nd_{sys} is omitted from writing.

8.3 Main results

In this section we consider SNPSP systems as acceptors or generators of sets of numbers in four modes: $maxs$, $maxps$, $mins$, and $minps$. It is known, e.g. from Chapter 7, that SNPSP systems as generators or acceptors are universal, operating in the "usual way" in membrane computing, i.e. neurons operate in parallel. Using the four modes, we show in this section that we can still achieve universality despite the restriction of induced sequentiality and having a normal form.

8.3.1 Sequential SNPSP systems based on max spike number

We first consider acceptors and generators in $maxs$ and $maxps$ modes: in $maxs$ mode, at most one neuron with the most stored spikes is nondeterministically chosen to fire; in $maxps$ mode, all neurons with the most number of spikes are allowed to fire. Our (non)universality results under a normal form are as follows (with details of the parameters provided shortly): for $maxs$, acceptors and generators are universal; for $maxps$, acceptors are universal while generators can become universal with nondeterminism provided in this work by nd_{syn}.

Max sequentiality ($maxs$) mode

Theorem 8.3.1. $NRE = N_{2,gen}SNPSP^{maxs}(rule_2, \pm syn_k, nd_{syn}), k \geq 1$.

Proof. It is enough to simulate a register machine M by means of an SNPSP system Π, with restrictions given in the Theorem statement. Before we construct Π, we provide a general description of the computation as follows: each register r in M is associated with a σ_r in Π. If r stores the value n, then σ_r stores $2n+2$ spikes. We will construct modules for Π that will simulate addition, subtraction, and halting operations in M. If a rule with label l_i is applied in M, then the associated neuron σ_{l_i} in Π is activated in order to simulate the operation performed by l_i. In the initial configuration, all neurons are empty, except neuron σ_{l_0} which contains one spike and is associated with the initial instruction l_0 of M.

Module ADD: The module simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ is shown in Figure 8.1. Once σ_{l_i} is activated it sends one spike each to $\sigma_{l_i^1}$ and $\sigma_{l_i^2}$. At this step $\sigma_{l_i^1}$ stores two spikes, while $\sigma_{l_i^2}$ stores one spike. Let step t be the step where $\sigma_{l_i^1}$ is activated: $\sigma_{l_i^1}$ applies its only rule so it consumes one spike and deletes the synapse (l_i^1, r) if it exists. At step $t+1$ we have the following: $\sigma_{l_i^1}$ becomes an active neuron to continue the application of its plasticity rule applied at t, while $\sigma_{l_i^2}$ is now the only activated neuron. At $t+1$, $\sigma_{l_i^2}$ sends one spike each to σ_r and σ_p, while $\sigma_{l_i^1}$ creates synapse (l_i^1, r), thus sending one spike to σ_r.

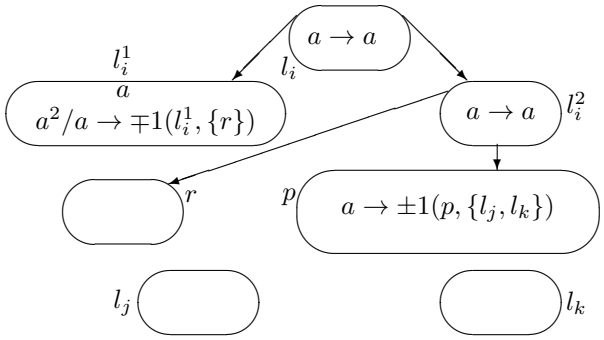

Figure 8.1: Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.1.

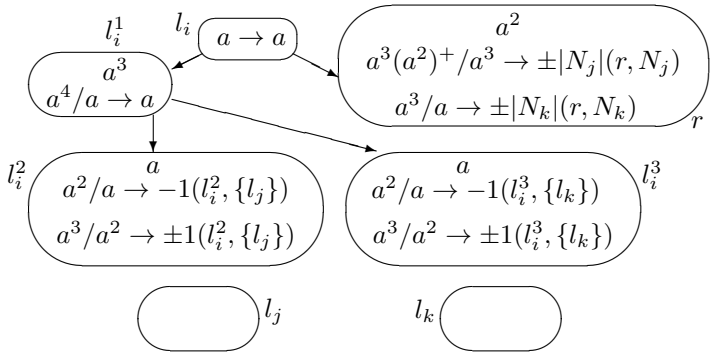

Figure 8.2: Module SUB simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$ in the proof of Theorem 8.3.1.

The spikes stored in σ_r at $t+1$ increase by two, and σ_p is activated at the next step. We omit further details of σ_r at this point, but we note that no rule of σ_r is activated while σ_r stores an even number of spikes. Since σ_p is activated, it will nondeterministically select whether to create synapse (p, l_j) or (p, l_k). Once σ_p selects one synapse to create, either σ_{l_j} or σ_{l_k} is activated. The ADD module correctly simulates the addition operation: the spikes stored in σ_r are increased by two, simulating the increase of the value stored in r by one; then, either σ_{l_j} or σ_{l_k} is activated nondeterministically, simulating the nondeterministic application of either l_j or l_k in M.

Module SUB: The module simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$ is shown in Figure 8.2. This is the only module in which σ_r becomes activated. We define N_k (N_j, resp.) as $N_k = \{l_i^3 | l_i \text{ is a } \texttt{SUB} \text{ instruction on } r\}$ ($N_j = \{l_i^2 | l_i \text{ is a } \texttt{SUB} \text{ instruction on } r\}$, resp.). Once activated, σ_{l_i} sends one spike each to $\sigma_{l_i^1}$ and σ_r. Let t be the step when σ_{l_i} and σ_r collect four and $2n+3$ spikes, respectively. At t we can have both $\sigma_{l_i^1}$ and σ_r activated, but due to *maxs* mode, only one neuron becomes activated. Depending on the value of n, either $\sigma_{l_i^1}$ or σ_r spikes, and we have the following two possible cases:

Case $n = 0$: This case corresponds to σ_r storing three spikes at t, so $\sigma_{l_i^1}$ becomes activated. In this case $\sigma_{l_i^1}$ consumes one spike and sends one spike each to $\sigma_{l_i^2}$ and $\sigma_{l_i^3}$ at t. At $t+1$, σ_r is the activated neuron with the most stored spikes. Recall that in this case M must deterministically

execute l_k, and this operation is simulated as follows: the rule of σ_r with $L(E) = \{a^3\}$ is applied and $|N_k|$ synapses are created and then deleted (in the next step). For the specific case where $N_k = \{l_i^3\}$ so that $|N_k| = 1$, σ_r consumes one spike and synapse (r, l_i^3) is created and then deleted. The spikes stored in σ_r return to two, simulating the "no operation" when register r stores $n = 0$. Now at $t+2$, $\sigma_{l_i^3}$ stores three spikes and applies its rule with $L(E) = \{a^3\}$, so synapse (l_i^3, l_k) is created and then deleted. Before σ_{l_k} is activated, *maxs* mode dictates that $\sigma_{l_i^2}$ must apply its rule with $L(E) = \{a^2\}$ to delete any synapse from it.

Case $n \geq 1$: This case corresponds to σ_r storing at least five spikes at t, so σ_r is the neuron that fires. In this case M must deterministically execute l_j which is simulated as follows: the rule of σ_r with $L(E) = \{a^3(a^2)^+\}$ is applied and $|N_j|$ synapses are created and then deleted (in the next step). For the specific case where $N_j = \{l_i^2\}$ so that $|N_j| = 1$, σ_r consumes three spikes and synapse (r, l_i^2) is created and then deleted. The stored spikes in σ_r after removing three spikes return to an even count, simulating the decrease of the value stored in r by one. At $t+1$, $\sigma_{l_i^1}$ is the activated neuron with the most stored spikes, and it sends one spike each to $\sigma_{l_i^2}$ and $\sigma_{l_i^3}$. At $t+2$, $\sigma_{l_i^2}$ now stores three spikes and applies its rule with $L(E) = \{a^2\}$, so synapse (l_i^2, l_j) is created and then deleted. Before σ_{l_j} is activated, $\sigma_{l_i^3}$ must apply its rule with $L(E) = \{a^2\}$ to delete any synapse from it.

In both cases, the number of stored spikes in $\sigma_{l_i^1}$, $\sigma_{l_i^2}$, and $\sigma_{l_i^3}$ are restored to the original number of spikes before σ_{l_i} was activated, i.e. the same SUB module can be used again. We consider now the general case where more than one $l_i : (\texttt{SUB}(r), l_j, l_k)$ operation is performed on the same register r. We need to be certain that there is no interference with several SUB modules acting on the same σ_r.

By definition of N_j and N_k, only the correct SUB module simulating l_x on r is allowed to continue because either l_x^2 or l_x^3 receives three spikes. The other neuron, including neurons l_y^2 and l_y^3 for l_y also operating on r, only receive two spikes: *maxs* mode will nondeterministically allow all such neurons to remove one spike from their two spikes using their plasticity rule with $L(E) = \{a^2\}$. Hence, the SUB module correctly simulates the subtraction operation: spikes stored in σ_r are reduced by two, simulating the decrease by one of the nonzero value stored in r, then σ_{l_j} is activated to simulate l_j in M; if r stored the value zero, then the two spikes in σ_r are restored and σ_{l_k} is activated to simulate l_k in M.

Module FIN: Once M arrives at l_h, M halts its computation and is simulated by the FIN module illustrated in Figure 8.3. Neuron l_h is activated, sending one spike to σ_{out}. Let t be the step when σ_{out} first spikes to the environment. At t, σ_{out} has six spikes and consumes two, reducing its spikes to four. At $t+1$, if register 1 stores the value $n = 1$, then σ_1 contains $2n + 3 = 5$ spikes. Due to *maxs* mode, both σ_{out} and σ_1 can be activated but only σ_1 is allowed to spike. At $t+1$ we have σ_1 reducing its spikes to three, and deleting the nonexisting synapse $(1, out)$. At $t+2$, σ_{out} spikes (because it is the only activated neuron) for the second and final time to the environment. The interval between the first and second spikes minus one is $((t+2) - t) - 1 = 1$, exactly the value

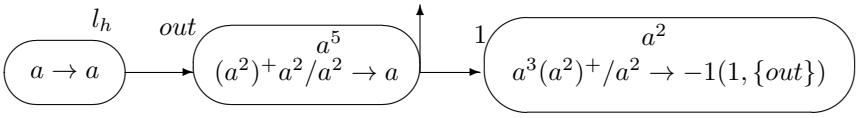

Figure 8.3: Module FIN in the proof of Theorem 8.3.1.

stored in register r. Note that the four spikes in σ_1 at $t+2$ do not allow σ_1 to become activated.

If register r stores $n \geq 2$, then σ_1 contains at least seven spikes. Again, σ_1 has more spikes than σ_{out}, and σ_1 reduces its spikes by two each step. The second spike of σ_{out} is produced at step $t+n+1$, so that the interval between the first and second spikes minus one is $((t+n+1)-t)-1 = n$, exactly the value stored in r. The Theorem parameters are satisfied in every neuron of every module: at most two rules, at most $k \geq 1$ synapses created (deleted), and nd_{sys} and nd_{syn} exist. This completes the proof. □

Remarks: The idea of an active neuron, used in our proofs, is from [41] and their activated neuron and implicit active neuron: In one step of their ADD module in $maxs$ mode, a neuron is activated while another neuron with delay is active. In our ADD module, we apply a plasticity rule instead of a rule with delay. If we further restrict the system and remove nd_{syn}, we still achieve universality with a minor trade-off: $k \geq 2$ instead of $k \geq 1$.

Theorem 8.3.2. $NRE = N_{2,gen}SNPSP^{maxs}(rule_2, \pm syn_k), k \geq 2$.

Proof. In the proof for Theorem 8.3.1, only the ADD module in Figure 8.1 has nd_{syn}. We use in this case the new ADD module in Figure 8.4, which is a modified version of the ADD module in Figure 8.1. Since only nd_{sys} remains, and σ_{p_1} and σ_{p_2} can be activated at the same step, only one of them will fire: if σ_{p_1} (σ_{p_2}, resp.) fires, using its plasticity rule with $L(E) = \{a\}$ it sends one spike to σ_{l_j} (σ_{l_k}, resp.) and σ_{p_2} (σ_{p_1}, resp.). Before σ_{l_j} (σ_{l_k}, resp.) is activated, σ_{p_2} (σ_{p_1}, resp.) uses its rule with $L(E) = \{a^2\}$ to remove its two spikes and delete synapse (p_2, r) ((p_1, r), resp.). Notice however that the synapse $(m, r), m \in \{p_1, p_2\}$ never exists, and r in this case can be replaced by any neuron that neuron m will never have a synapse to. All Theorem parameters are satisfied: only nd_{sys} exist, and in this case $k \geq 2$ from the new ADD module. □

Theorem 8.3.3. $NRE = N_{acc}SNPSP^{maxs}(rule_2, \pm syn_k), k \geq 1$.

Proof. Since a register machine M working as an acceptor characterizes NRE using only addition instructions of the form $l_i : (\texttt{ADD}(r), l_j)$, we modify the ADD module in Figure 8.4 as follows: we remove σ_{p_1}, σ_{p_2}, σ_{l_k}, and add synapse (l_i^2, l_j); We store $2n+2$ spikes in σ_1, associated with n (the value to be accepted or rejected) stored in register 1 of M. We use the SUB module in Figure 8.2, and $k \geq 1$ instead of $k \geq 2$. The FIN module is not required: we accept n if σ_{l_h} is activated (meaning M has reached l_h) and reject n otherwise. □

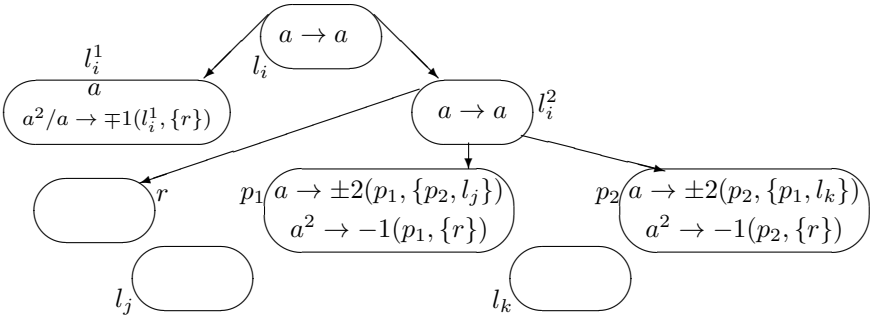

Figure 8.4: Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.2.

Max pseudo-sequentiality (*maxps*) mode

Theorem 8.3.4. $NRE = N_{2,gen}SNPSP^{maxps}(rule_2, \pm syn_k, nd_{syn}), k \geq 1$.

Proof. All modules in the proof of Theorem 8.3.1 can be used in *maxps* mode. In the SUB module in Figure 8.2 in particular, if $\sigma_{l_i^2}$ ($\sigma_{l_i^3}$, resp.) has three spikes, then $\sigma_{l_i^3}$ ($\sigma_{l_i^2}$, resp.) and every neuron with a label in $N_j - \{l_i^2\}$ ($N_k - \{l_i^3\}$, resp.) only have two spikes: these neurons with two spikes remove their spikes in parallel by applying their rule with $L(E) = \{a^2\}$. □

Remarks: In the case of deterministic generators, without a source of nondeterminism, they only generate singleton sets given an initial configuration (hence they are not universal) i.e. $NRE \neq D_{gen}SNPSP^{maxps}(rule_*, \pm syn_*)$. Similar to deterministic register machines as acceptors, we can have universality if the previous result is considered using SNPSP systems as acceptors instead.

Theorem 8.3.5. $NRE = D_{acc}SNPSP^{maxps}(rule_2, \pm syn_k), k \geq 1$.

Proof. This result holds since all modules in Theorem 8.3.3 are deterministic. □

8.3.2 Sequential SNPSP systems based on min spike number

Next we consider acceptors and generators in *mins* and *minps* modes: *mins* mode dictates that at most one neuron (with the least number of spikes stored) is nondeterministically chosen to become activated step; for *minps* mode, all the neurons with the least number of spikes stored become activated. Our (non)universality results under a normal form are as follows (with details of the parameters provided shortly): acceptor and generator systems in *mins* mode are universal; generator systems in *minps* mode are not universal; generator systems with nd_{syn}, or acceptor systems in *minps* mode, are universal.

Min sequentiality (*mins*) mode

Theorem 8.3.6. $NRE = N_{2,gen}SNPSP^{mins}(rule_2, \pm syn_k), k \geq 2$.

Proof. We construct modules of an SNPSP system Π which simulates a register machine M. Modules of Π will again perform addition, subtraction, and halting operations corresponding to the same operations in M. If register r stores the value n, then σ_r stores $2n+2$ spikes. Simulating instruction l_i means σ_{l_i} in a module is activated, so that the module performs the operation of l_i.

Module ADD: The ADD module is illustrated in Figure 8.5. After σ_{l_i} becomes activated, it sends one spike each to σ_r and $\sigma_{l_i^1}$. Due to *mins* mode, only $\sigma_{l_i^1}$ is allowed by the system to apply a rule because it has only one spike (σ_r has at least three spikes). After $\sigma_{l_i^1}$ spikes, one spike is sent to σ_r, σ_{p_1}, and σ_{p_2}. In this way, two spikes are added to σ_r simulating the increment in register r. The *mins* mode will either allow σ_{p_1} or σ_{p_2} to become activated first: if σ_{p_1} (σ_{p_2}, resp.) becomes activated first, it creates two synapses so that σ_{l_j} and σ_{p_2} (σ_{l_k} and σ_{p_1}, resp.) receive one spike each; Before σ_{l_j} (σ_{l_k}, resp.) is allowed to apply a rule, *mins* mode dictates that σ_{p_2} (σ_{p_1}, resp.) apply its rule first because σ_{p_2} (σ_{p_1}, resp.) only has two spikes. Once σ_{p_2} (σ_{p_1}, resp.) removes its two spikes, σ_{l_j} (σ_{l_k}, resp.) can proceed to simulating instruction l_j (l_k, resp.). The `ADD` instruction is correctly simulated in Π.

Module SUB: The SUB module is illustrated in Figure 8.6. After σ_{l_i} spikes, $\sigma_{l_i^1}$ and σ_r each contain two and $2n+3$ spikes, respectively. Again, *mins* mode allows $\sigma_{l_i^1}$ to spike before σ_r. Let t be the step that $\sigma_{l_i^1}$ spikes, i.e. the step when $\sigma_{l_i^1}$ becomes activated: at t, $\sigma_{l_i^1}$ deletes all synapses from itself; at $t+1$, $\sigma_{l_i^1}$ remains active, then creates synapses and sends one spike each to $\sigma_{l_i^2}$ and $\sigma_{l_i^3}$; Also at $t+1$, σ_r becomes activated. As in Theorem 8.3.1, we define the sets N_j and N_k as follows: $N_k = \{l_i^3 | l_i \text{ is a } \texttt{SUB} \text{ instruction on } r\}$ ($N_j = \{l_i^2 | l_i \text{ is a } \texttt{SUB} \text{ instruction on } r\}$, resp.).

If $r \geq 1$ ($r = 0$, resp.), then σ_r creates a synapse and sends one spike each to every neuron with a label in N_j (N_k, resp.) at $t+1$. Also, if $r = 0$ then σ_r will contain $3-1=2$ spikes again, otherwise if $r \geq 1$ then σ_r will contain $2n+3-3=2n$ spikes again. At $t+2$, $\sigma_{l_i^2}$ ($\sigma_{l_i^3}$, resp.) contains two spikes, while $\sigma_{l_i^3}$ and every neuron with a label in $N_j - \{l_i^2\}$ ($\sigma_{l_i^2}$ and every neuron with a label in $N_k - \{l_i^3\}$, resp.) only contain one spike. At $t+2$, *mins* mode dictates that these neurons containing one spike must remove their spikes, using their rule with $\alpha := -$. At $t+3$, σ_{l_j} (σ_{l_k}, resp.) is activated by $\sigma_{l_i^2}$ ($\sigma_{l_i^3}$, resp.) because it now contains three spikes. The `SUB` instruction is correctly simulated in Π, and no interference occurs among several `SUB` instructions operating on the same register r.

Module FIN: The FIN module is illustrated in Figure 8.7. Once σ_{l_h} is activated, σ_{out} then contains four spikes. At the next step, σ_{out} is activated and sends the first spike to the environment (we denote this step as t). At $t+1$, σ_1 contains at least five spikes, since $n \geq 1$. It takes $n-1$ steps for σ_1, using its rule with $\alpha := -$, to reduce its spikes to five if $n \geq 2$. Once σ_1 stores five spikes, it creates a synapse and sends one spike to σ_{out} at $t+n$. At $t+n+1$, σ_{out} sends a spike to the environment for the second and final time. The value $((t+n+1)-t)-1) = n$ is computed, which is precisely the value stored in register 1 of M. Note that the second spike sent by σ_{out} to σ_1 never activates any rule in σ_1.

Clearly, only nd_{sys} exists, at most two rules exist in each neuron, and at least $k = 2$ synapses

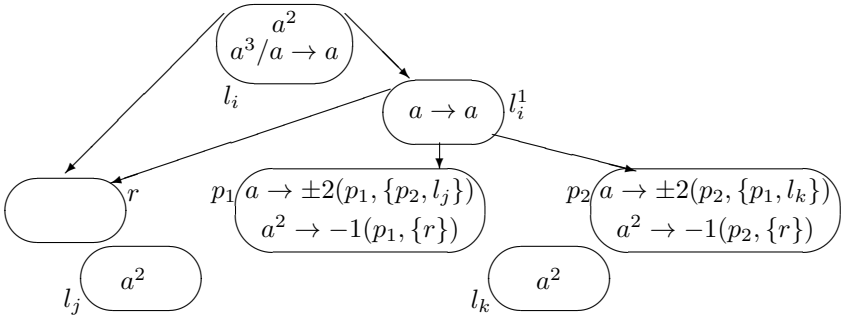

Figure 8.5: Module ADD simulating $l_i : (\texttt{ADD}(r), l_j, l_k)$ in the proof of Theorem 8.3.6.

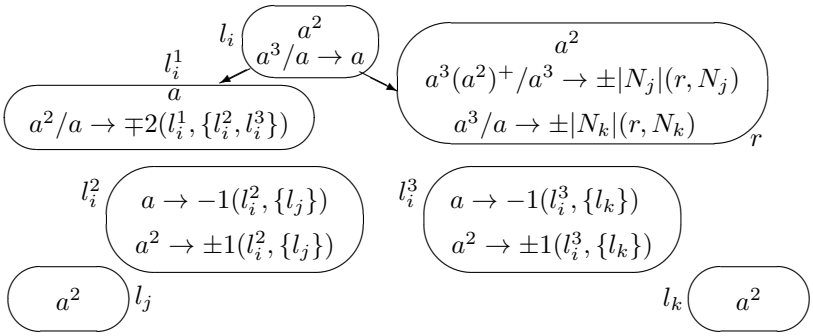

Figure 8.6: Module SUB simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$ in the proof of Theorem 8.3.6.

are created each step. This completes the proof. □

Theorem 8.3.7. $NRE = N_{acc}SNPSP^{mins}(rule_2, \pm syn_k), k \geq 2$.

Proof. To simulate an acceptor register machine M using addition instructions of the form $l_i : (\texttt{ADD}(r), l_j)$, we modify the ADD module in Figure 8.5 as follows: we remove σ_{p_1}, σ_{p_2}, σ_{l_k}, and add synapse (l_i^1, l_j). We store $2n + 2$ spikes in σ_1, associated with the value n in register 1 of M. The SUB module is the module in Figure 8.6, and a FIN module is not required. We accept n if σ_{l_h} becomes activated, and reject n otherwise. □

Min pseudo-sequentiality (*minps*) mode

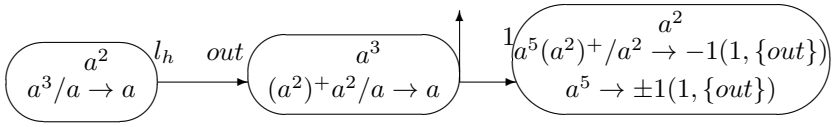

Figure 8.7: Module FIN in the proof of Theorem 8.3.6.

Without a source of nondeterminism, deterministic SNPSP system as generators in *minps* mode are not universal, i.e. $NRE \neq D_{gen}SNPSP^{minps}(rule_*, \pm syn_*)$. This nonuniversality parallels the nonuniversality of deterministic generators in *maxps* mode. As in Theorem 8.3.4 for generators in *maxps* mode, generators in *minps* mode can become universal using nd_{syn}, as the next result shows.

Theorem 8.3.8. $NRE = N_{2,gen}SNPSP^{minps}(rule_2, \pm syn_k, nd_{syn}), k \geq 2$.

Proof. We modify the ADD module in Figure 8.5, to remove nd_{sys} and maintain only nd_{syn}, as follows: Replace σ_{p_1} and σ_{p_2} with a single neuron σ_p, and add synapse (l_i^1, p); the rule set of σ_p is $R_p = \{a \to \pm 1(p, \{l_j, l_k\})\}$.

The SUB and FIN modules in Figure 8.6 and 8.7, respectively, can be used in *minps* mode. In particular, the SUB module is still correct because if $\sigma_{l_i^2}$ ($\sigma_{l_i^3}$, resp.) contains two spikes, then $\sigma_{l_i^3}$ and every neuron with a label in $N_j - \{l_i^2\}$ ($\sigma_{l_i^2}$ and every neuron with a label in $N_k - \{l_i^3\}$, resp.) contain one spike each. The neurons that contain one spike each must remove their spikes in parallel, before the next instruction is correctly simulated. □

Theorem 8.3.9. $NRE = N_{acc}SNPSP^{minps}(rule_2, \pm syn_k), k \geq 2$.

Proof. All modules in the proof of Theorem 8.3.7 can be used in *minps* mode. □

Chapter 9

Asynchronous SNPSP systems

In this chapter we consider SNP systems with structural plasticity (in short, SNPSP systems) working in the asynchronous (in short, *asyn*) mode. We prove that for *asyn* mode, bounded SNPSP systems are not universal, while unbounded SNPSP systems with weighted synapses are universal. Our results provide support to the conjecture of the still open problem of whether asynchronous SNP systems with standard rules are universal.

9.1 Introduction

The restriction we apply to SNPSP systems in this chapter is asynchronous operation: imposing synchronization on biological functions is sometimes "too much", i.e. not always realistic. Hence, the asynchronous mode of operation is interesting to consider from a bio-inspired perspective. Asynchronous restriction is also interesting computationally, since we can remove the global and synchronizing clock which computing units must follow. We refer the readers again to [18], [39], and [89] for further bio-inspired and computing motivations of asynchronous operation. In this chapter we prove that (i) asynchronous bounded (i.e. there exists a bound on the number of stored spikes in any neuron) SNPSP systems are not universal, (ii) asynchronous weighted (i.e. a positive integer weight is associated with each synapse) SNPSP systems, even under a normal form (provided below), are universal. The open problem in [18] whether asynchronous bounded SNP systems with standard rules are universal is conjectured to be false. Also, asynchronous SNP systems with extended rules are known to be universal [19]. Our results provide some support to the conjecture, since neurons in SNPSP systems produce at most one spike each step (similar to standard rules) while synapses with weights function similar to extended rules (more than one spike can be produced each step).

9.2 Asynchronous SNPSP systems

From Chapter 8 we can have the following nondeterminism levels: *rule-level* and *synapse-level*. There is also the system-level but this level is not relevant in this chapter. In this chapter however

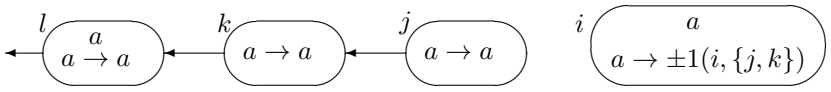

Figure 9.1: An SNPSP system Π_{ej}.

we introduce another level, given by the asynchronous operation of the system: *neuron-level*, if at least one activated neuron with rule r can choose to apply its rule r or not (i.e. asynchronous). By default SNP and SNPSP systems are locally sequential (at most one rule is applied per neuron) but globally parallel (all activated neurons must apply a rule). The application of rules in neurons is usually synchronous, i.e. a global clock is assumed and if a neuron can apply a rule then it must do so. However, in the asynchronous (*asyn*, in short) mode we release this synchronization so that neuron-level nondeterminism is implied.

A result of a computation in *asyn* mode cannot be the time interval between two spikes, as there is no "schedule" or theoretical limit to when neurons must apply their rule. For SNPSP systems in *asyn* mode and as in [18][19][89], the output is obtained by counting the total number of spikes sent out by σ_{out} to the environment upon reaching a halting configuration. We refer to Π as generator, if Π computes in this asynchronous manner. The accepting mode is not considered in this chapter.

For our universality results, the following simplifying features are used in our systems as the normal form: (i) plasticity rules can only be found in purely plastic neurons (i.e. neurons with plasticity rules only), (ii) neurons with standard rules are simple, and (iii) we do not use forgetting rules or rules with delays. We denote the family of sets computed by asynchronous SNPSP systems (under the mentioned normal form) as generators as $N_{tot}SNPSP^{asyn}$: subscript *tot* indicates the total number of spikes sent to the environment as the result; Other parameters are as follows: $+syn_k$ ($-syn_j$, respectively) where at most k (j, resp.) synapses are created (deleted, resp.) each step; $nd_\beta, \beta \in \{syn, rule, neur\}$ indicate additional levels of nondeterminism source; $rule_m$ indicates at most m rules (either standard or plasticity) per neuron; Since our results for k and j for $+syn_k$ and $-syn_j$ are equal, we write them instead in the compressed form $\pm syn_k$, where $\pm$ in this sense is not the same as when $\alpha = \pm$. A bound p on the number of spikes stored in any neuron of the system is denoted as $bound_p$. We omit nd_{neur} from writing since it is implied in *asyn* mode.

To illustrate the notions and semantics of asynchronous SNPSP systems, we take as an example the SNPSP system Π_{ej} of degree 4 in Fig. 9.1, and describe its computations. The initial configuration is as follows: spike distribution is $\langle 1, 0, 0, 1\rangle$ for the neuron order σ_i, σ_j, σ_k, σ_l, respectively; $syn = \{(j, k), (k, l)\}$; output neuron is σ_l, indicated by the outgoing synapse to the environment.

Given the initial configuration, σ_i and σ_l can become activated. Due to *asyn* mode however, they can decide to apply their rules at a later step. If σ_l applies its rule before it receives a spike from σ_i, then it will spike to the environment twice so that $N_{tot}(\Pi_{ej}) = \{2\}$. Since $k = 1 < |\{j, k\}|$ and $pres(i) = \emptyset$, σ_i nondeterministically selects whether to create synapse (i, j) or (i, k); if (i, j) ((i, k), resp.) is created; a spike is sent from σ_i to σ_j (σ_k, resp.) due to the embedded sending of

a spike during synapse creation. Let this be step t. If (i, j) is created then $syn' = syn \cup \{(i, j)\}$, otherwise $syn'' = syn \cup \{(i, k)\}$. At $t + 1$, σ_i deletes the created synapse at t (since $\alpha = \pm$), and we have syn again. Note that if σ_l does not apply its rule and collects two spikes (one spike from σ_i), the computation is aborted or blocked, i.e. no output is produced since $a^2 \notin L(a)$.

9.3 Main results

In this section we use at most two nondeterminism sources: nd_{neur} (in $asyn$ mode), and nd_{syn}. Recall that in $asyn$ mode, if σ_i is activated at step t so that an $r \in R_i$ can be applied, σ_i can choose to apply r or not. If σ_i did not choose to apply r, σ_i can continue to receive spikes so that for some $t' > t$, it is possible that: r can never be applied again, or some $r' \in R_i, r' \neq r$, is applied. For the next result, each neuron can store only a bounded number of spikes (see for example [18][39][44] and references therein). In [39], it is known that bounded SNP systems with extended rules in $asyn$ mode characterize $SLIN$, but it is open whether such result holds for systems with standard rules only. In [18], a negative answer was conjectured for the following open problem: are asynchronous SNP systems with standard rules universal?

Lemma 9.3.1. $N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}) \subseteq SLIN, p \geq 1$.

Proof. Taking any asynchronous SNPSP system Π with a given bound p on the number of spikes stored in any neuron, we observe that the number of possible configurations is finite: Π has a constant number of neurons, and that the number of spikes stored in each neuron are bounded. We then construct a right-linear grammar G, such that Π generates the length set of the regular language $L(G)$. Let us denote by $\mathcal{C}$ the set of all possible configurations of Π, with C_0 being the initial configuration. The right-linear grammar $G = (\mathcal{C}, \{a\}, C_0, P)$, where the production rules in P are as follows:

(1) $C \to C'$, for $C, C' \in \mathcal{C}$ if Π has a transition $C \Rightarrow C'$ in which the output neuron does not spike;

(2) $C \to aC'$, for $C, C' \in \mathcal{C}$ if Π has a transition $C \Rightarrow C'$ in which the output neuron spikes;

(3) $C \to \lambda$, for any $C \in \mathcal{C}$ in which Π halts.

Due to the construction of G, Π generates the length set of $L(G)$, hence the set is semilinear. □

Lemma 9.3.2. $SLIN \subseteq N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}), p \geq 1$.

The proof is based on the following observation: A set Q is semilinear if and only if Q is generated by a strongly monotonic register machine M. It suffices to construct an SNPSP system Π with restrictions given in the theorem statement, such that Π simulates M. Recall that M has precisely register 1 only (it is also the output register) and addition instructions of the form $l_i : (\texttt{ADD}(1), l_j, l_k)$.

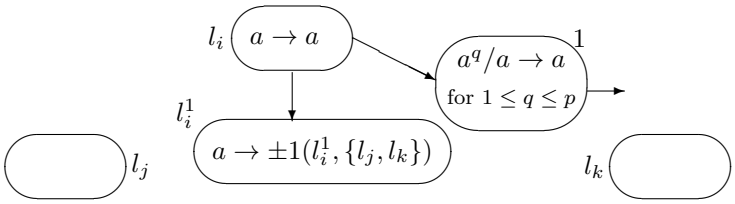

Figure 9.2: Module ADD simulating $l_i : (\texttt{ADD}(1) : l_j, l_k)$ in the proof of Lemma 9.3.2.

The ADD module for Π is given in Fig. 9.2. Next, we describe the computations in Π. Once ADD instruction l_i of M is applied, σ_{l_i} is activated and it sends one spike each to σ_1 and $\sigma_{l_i^1}$. At this point we have two possible cases due to *asyn* mode: either σ_1 spikes to the environment before $\sigma_{l_i^1}$ spikes, or after. If σ_1 spikes before $\sigma_{l_i^1}$, then the number of spikes in the environment is immediately incremented by one. After some time, the computation will proceed if $\sigma_{l_i^1}$ applies its only (plasticity) rule. Once $\sigma_{l_i^1}$ applies its rule, either σ_{l_j} or σ_{l_k} becomes nondeterministically activated.

However, if σ_1 spikes after $\sigma_{l_i^1}$ spikes, then the number of spikes in the environment is not immediately incremented by one since σ_1 does not consume a spike and fire to the environment. The next instruction, either l_j or l_k, is then simulated by Π. Furthermore, due to *asyn* mode, the following "worst case" computation is possible: σ_{l_h} becomes activated (corresponding to l_h in M being applied, thus halting M) before σ_1 spikes. In this computation, M has halted and has applied an m number of ADD instructions since the application of l_i. Without loss of generality we can have the arbitrary bound $p > m$, for some positive integer p. We then have the output neuron σ_1 storing m spikes. Since the rules in σ_1 are of the form $a^q/a \to a$, $1 \leq q \leq p$, σ_1 consumes one spike at each step it decides to apply a rule, starting with rule $a^m/a \to a$, until rule $a \to a$. Thus, Π will only halt once σ_1 has emptied all spikes it stores, sending m spikes to the environment in the process.

The FIN module is not necessary, and we add σ_{l_h} without any rule (or maintain $pres(l_h) = \emptyset$). Once M halts by reaching instruction l_h, a spike in Π is sent to neuron l_h. Π is clearly bounded: every neuron in Π can only store at most p spikes, at any step. We then have Π correctly simulating the strongly monotonic register machine M. This completes the proof. □

From Lemma 9.3.1 and 9.3.2, we can have the next result.

Theorem 9.3.1. $SLIN = N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}), p \geq 1$.

Next, in order to achieve universality, we add an additional ingredient to asynchronous SNPSP systems: weighted synapses. The ingredient of weighted synapses has already been introduced in SNP systems literature, and we refer the reader to [96] (and references therein) for computing and biological motivations. In particular, if σ_i applies a rule $E/a^c \to a^p$, and the weighted synapse (i, j, r) exists (i.e. the weight of synapse (i, j) is r) then σ_j receives $p \times r$ spikes. A natural consideration is weighted synapses for asynchronous SNPSP systems: since asynchronous SNPSP systems are not universal, we look for other ways to improve their power. SNPSP systems with weighted synapses (in

short, WSNPSP systems) are defined in a similar way as SNPSP systems, except for the plasticity rules and *syn*. Plasticity rules in σ_i are now of the form

$$E/a^c \rightarrow \alpha k(i, N, r),$$

where $r \geq 1$, and E, c, α, k, N are as previously defined. Every synapse created by σ_i using a plasticity rule with weight r receives the weight r. Instead of one spike sent from σ_i to a σ_j during synapse creation, $j \in N$, r spikes are sent to σ_j. The synapse set is now of the form

$$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\} \times \mathbb{N}^+.$$

We note that SNPSP systems are special cases of SNPSP systems with weighted synapses where $r = 1$, and when $r = 1$ we omit it from writing. In weighted SNP systems with standard rules, the weights can allow neurons to produce more than one spike each step, similar to having extended rules. In this way, our next result parallels the result that asynchronous SNP systems with extended rules are universal in [19]. However, our next result uses nd_{syn} with *asyn* mode, while in [19] their systems use nd_{rule} with *asyn* mode. We also add the additional parameter l in our universality result, where the synapse weight in the system is at most l. Our universality result also makes use of the normal form given in Section 9.2.

Theorem 9.3.2. $N_{tot}WSNPSP^{asyn}(rule_m, \pm syn_k, weight_l, nd_{syn}) = NRE, m \geq 9, k \geq 1, l \geq 3.$

Proof. We construct an asynchronous SNPSP system with weighted synapses Π, with restrictions given in the theorem statement, to simulate a register machine M. The simulation description is as follows: each register r of M corresponds to σ_r in Π. If register r stores the value n, σ_r stores $2n$ spikes. Simulating instruction $l_i : (\texttt{OP}(r) : l_j, l_k)$ of M in Π corresponds to σ_{l_i} becoming activated. After σ_{l_i} is activated, the operation OP is performed on σ_r, and either σ_{l_j} or σ_{l_k} becomes activated.

Module ADD: The module is shown in Fig. 9.3. At some step t, σ_{l_i} sends a spike to $\sigma_{l_i^1}$. At some $t' > t$, $\sigma_{l_i^1}$ sends a spike: the spike sent to σ_r is multiplied by two, while one spike is received by $\sigma_{l_i^2}$. For now we omit further details for σ_r, since it is never activated with an even number of spikes. At some $t'' > t'$, $\sigma_{l_i^2}$ nondeterministically creates (then deletes) either (l_i^2, l_j) or (l_i^2, l_k). The chosen synapse then allows either σ_{l_j} or σ_{l_k} to become activated. The ADD module thus increments the contents of σ_r by two, simulating the increment by one of register r. Next, only one among σ_{l_j} or σ_{l_k} becomes nondeterministically activated. The addition operation is correctly simulated.

Module SUB: The module is shown in Fig. 9.4. Let $|S_r|$ be the number of instructions with form $l_i : (\texttt{SUB}(r), l_j, l_k)$, and $1 \leq s \leq |S_r|$. $|S_r|$ is the number of SUB instructions operating on register r, and we explain in a moment why we use a size of a set for this number. Clearly, when no SUB operation is performed on r, then $|S_r| = 0$, as in the case of register 1. At some step t, σ_{l_i} spikes, sending 1 spike to σ_r, and $4|S_r| - s$ spikes to $\sigma_{l_i^1}$ (the weight of synapse (l_i, l_i^1)).

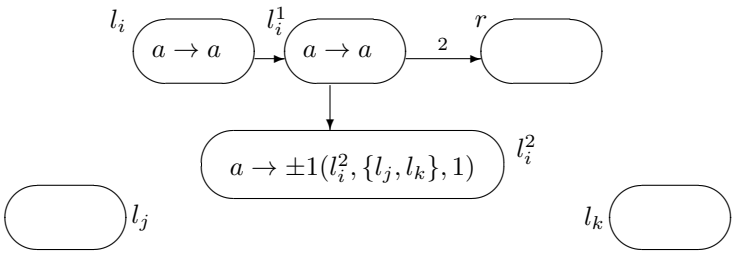

Figure 9.3: Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$ in the proof of Theorem 9.3.2.

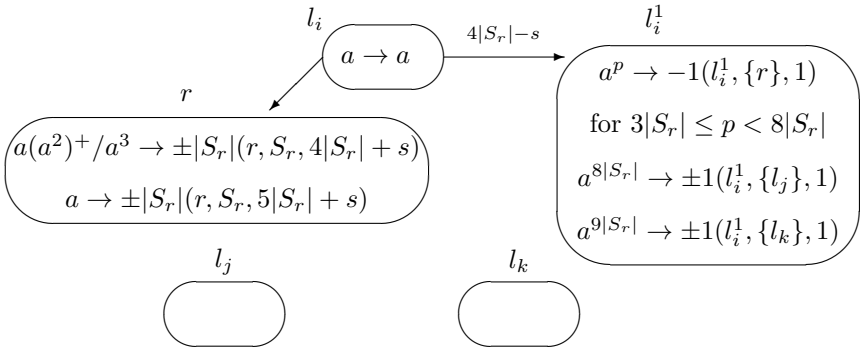

Figure 9.4: Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$ in the proof of Theorem 9.3.2.

$\sigma_{l_i^1}$ has rules of the form $a^p \to -1(l_i^1, \{r\}, 1)$, for $3|S_r| \leq p < 8|S_r|$. When one of these rules is applied, it performs similar to a forgetting rule: p spikes are consumed and deletes a nonexisting synapse (l_i^1, r). Since $\sigma_{l_i^1}$ received $4|S_r| - s$ spikes from σ_{l_i}, and $3|S_r| \leq 4|S_r| - s < 8|S_r|$, then one of these rules can be applied. If $\sigma_{l_i^1}$ applies one of these rules at $t' > t$, no spike remains. Otherwise, the $4|S_r| - s$ spikes can combine with the spikes from σ_r at a later step. In the case where register r stores $n = 0$ (respectively, $n \geq 1$), then instruction l_k (respectively, l_j) is applied next. This case corresponds to σ_r applying the rule with $L(E) = \{a\}$ (respectively, $L(E) = \{a(a^2)^+\}$), which at some later step allows σ_{l_k} (respectively, σ_{l_j}) to be activated.

For the moment let us simply define $S_r = \{l_i^1\}$ (the general case is provided shortly). For case $n = 0$ (respectively, $n \geq 1$), σ_r stores 0 spikes (respectively, at least two spikes), so that at some $t'' > t$ the synapse $(r, l_i^1, 5|S_r| + s)$ (respectively, $(r, l_i^1, 4|S_r| + s)$) is created and then deleted. $\sigma_{l_i^1}$ then receives $5|S_r| + s$ spikes (respectively, $4|S_r| + s$ spikes) from σ_r. Note that we can have $t'' \geq t'$ or $t'' \leq t'$, due to *asyn* mode, where t' is again the step that $\sigma_{l_i^1}$ applies a rule. If $\sigma_{l_i^1}$ previously removed all of its spikes using its rules with $L(E) = \{a^p\}$, then it again removes all spikes from σ_r because $3|S_r| \leq x < 8|S_r|$, where $x \in \{4|S_r| + s, 5|S_r| + s\}$. At this point, no further rules can be applied, and the computation aborts, i.e. no output is produced. If however $\sigma_{l_i^1}$ did not remove its spikes previously, then it collects a total of either $8|S_r|$ or $9|S_r|$ spikes. Either σ_{l_j} or σ_{l_k} is then activated by $\sigma_{l_i^1}$ at a step after t''.

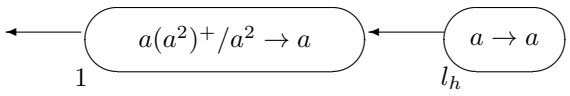

Figure 9.5: Module FIN in the proof of Theorem 9.3.2.

To remove the possibility of "wrong" simulations when at least two SUB instructions operate on register r, we give the general definition of S_r: $S_r = \{l_v^1 | l_v \text{ is a SUB instruction on register } r\}$. In the SUB module, a rule application in σ_r creates (and then deletes) an $|S_r|$ number of synapses: one synapse from σ_r to all neurons with label $l_v^1 \in S_r$. Again, each neuron with label l_v^1 can receive either $4|S_r| + s$, or $5|S_r| + s$ spikes from σ_r, and $4|S_r| - s$ spikes from σ_{l_v}.

Let l_i be the SUB instruction that is currently being simulated in Π. In order for the correct computation to continue, only $\sigma_{l_i^1}$ must not apply a rule with $L(E) = \{a^p\}$, i.e. it must not remove any spikes from σ_r or σ_{l_i}. The remaining $|S_r| - 1$ neurons of the form l_v^1 must apply their rules with $L(E) = \{a^p\}$ and remove the spikes from σ_r. Due to *asyn* mode, the $|S_r| - 1$ neurons can choose not to remove the spikes from σ_r: these neurons can then receive further spikes from σ_r in future steps, in particular they receive either $4|S_r| + s'$ or $5|S_r| + s'$ spikes, for $1 \leq s' \leq S_r$; these neurons then accumulate a number of spikes greater than $8|S_r|$ (hence, no rule with $E = a^p$ can be applied), but not equal to $8|S_r|$ or $9|S_r|$ (hence, no plasticity rule can be applied). Similarly, if these spikes are not removed, and spikes from $\sigma_{l_{v'}}$ are received, $v \neq v'$ and $l_{v'} \in S_r$, no rule can again be applied: if $l_{v'}$ is the s'th SUB instruction operating on register r, then $s \neq s'$ and $\sigma_{l_{v'}}$ accumulates a number of spikes greater than $8|S_r|$ (the synapse weight of $(l_{v'}, l_{v'}^1)$ is $4|S_r| - s'$), but not equal to $8|S_r|$ or $9|S_r|$. No computation can continue if the $|S_r| - 1$ neurons do not remove their spikes from σ_r, so computation aborts and no output is produced, i.e. only the computations in Π that are allowed to continue are the computations that correctly simulate a SUB instruction in M.

The SUB module correctly simulates a SUB instruction: instruction l_j is simulated only if r stores a positive value (after decrementing by one the value of r), otherwise instruction l_k is simulated (the value of r is not decremented).

Module FIN: The module FIN for halting the computation of Π is shown in Fig. 9.5. The operation of the module is clear: once M reaches instruction l_h and halts, σ_{l_h} becomes activated. Neuron l_h sends a spike to σ_1, the neuron corresponding to register 1 of M. Once the number of spikes in σ_1 become odd (of the form $2n + 1$, where n is the value stored in register 1), σ_1 keeps applying its only rule: at every step, 2 spikes are consumed, and 1 spike is sent to the environment. In this way, the number n is computed since σ_1 will send precisely n spikes to the environment.

The ADD module has nd_{syn}: initially it has $pres(l_i^2) = \emptyset$, and its $k = 1 < |N|$. We also observe the parameter values: m is at least 9 by setting $|S_r| = 1$, then adding the two additional rules in $\sigma_{l_i^1}$; k is clearly at least 1; lastly, the synapse weight l is at least 3 by again setting $|S_r| = 1$. This completes the proof. □

Chapter 10

Solving `Subset Sum` using SNPSP systems

In this chapter we provide solutions, a semi-uniform and a uniform one, to the numerical **NP**-complete problem `Subset Sum` using SNPSP systems. Both solutions are nondeterministic, with a normal form: using only standard spiking rules (without delays) and plasticity rules, and with synapse-level nondeterminism only. Also, all solutions are performed in a constant time, with the trade-off being space in terms of the number of neurons in the system. The nondeterminism introduced by plasticity rules in SNPSP systems can be used to decrease the number of neurons needed when there is a need to nondeterministically generate assignment of variables.

10.1 Introduction

As mentioned in Chapter 4, SNP systems have been used to solve hard problems. These solutions often fall into three categories: a *uniform* solution means that we construct only one solution (in this case, a system) for any instance of the problem; a *non-uniform* solution is one that depends on specific instances of a problem; a *semi-uniform* is one that is non-uniform and the construction is done in polynomial time. In what follows, we provide some details regarding problems solvable by SNPSP systems, based on [22][51][52][69][81][84][95]. Uniform solutions are preferred over non-uniform solutions since the former implies that the solution is based on the structure or features of the problem instead of specific instances of the problem. However, and since we assume $\mathbf{P} \neq \mathbf{NP}$, uniform solutions can come with a price of either increasing computation time or space, compared to non-uniform solutions. For non-uniform solutions, input neurons are not needed since the problem instance is embedded in the system (e.g. number of spikes or neurons, rules) while in uniform solutions, at least one input neuron is needed to introduce the instance into the system.

More formally, let $X = (I_X, \Theta_X)$ be a decision problem, and $g : \mathbb{N} \to \mathbb{N}$ a computable function, where I_X is a set of instances and Θ_X is a predicate over I_X. We say X is solvable by a family $\mathbf{\Pi} = \{\Pi(n) | n \in \mathbb{N}\}$ of SNPSP systems, in time bounded by g, in a *nondeterministic* and *uniform way* if the following hold:

- The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines, i.e. there exists a deterministic Turing machine working in polynomial time which constructs $\Pi(n), n \in \mathbb{N}$.
- There exist polynomial time computable functions, *cod* and s, over I_X, such that
  - For each instance $w \in I_X$, $s(w)$ is a natural number, and $cod(w)$ is a valid input (using some encoding) of the SNPSP system $\Pi(s(w))$.
  - The family $\mathbf{\Pi}$ is g-bounded with respect to (X, cod, s), i.e. for each instance $w \in I_X$, the minimum length of an accepting computation of $\Pi(s(w))$ with input $cod(w)$ is bounded by $g(|w|)$.
  - The family $\mathbf{\Pi}$ is *sound* with respect to (X, cod, s), i.e. for each $w \in I_X$, if there exists an accepting computation of $\Pi(s(w))$ with input $cod(w)$, then $\Theta_X(w) = 1$.
  - The family $\mathbf{\Pi}$ is *complete* with respect to (X, cod, s), i.e. for every $w \in I_X$, if $\Theta_X(w) = 1$ then there exists a computation of $\Pi(w)$ with input $cod(w)$ which is an accepting computation.

We say $X = (I_X, \Theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) | n \in \mathbb{N}\}$ of SNPSP systems, in a nondeterministic and uniform way if there exists a $k \in \mathbb{N}$ such that X is solvable by the family $\mathbf{\Pi}$ in time bounded by a polynomial, in a nondeterministic and uniform way. Now let $X = (I_X, \Theta_X)$ be a decision problem, and g, I_X, and Θ_X are as previously defined here. We say X is solvable by a family $\mathbf{\Pi} = \{\Pi(w) | w \in I_X\}$ of SNPSP systems, in time bounded by g, in a *nondeterministic* and *semi-uniform way* if the following hold:

- The family $\mathbf{\Pi}$ is polynomially uniform Turing machines, i.e. there exists a deterministic Turing machine working in polynomial time which constructs $\Pi(w)$ from instance $w \in I_X$.
- The family $\mathbf{\Pi}$ is g-bounded with respect to X, i.e. for each $w \in I_X$ the minimum length of an accepting computation of $\Pi(w)$ is bounded by $g(|w|)$.
- The family $\mathbf{\Pi}$ is *sound* with respect to X, i.e. for every $w \in I_X$, if there exists an accepting computation of $\Pi(w)$, then $\Theta_X(w) = 1$.
- The family $\mathbf{\Pi}$ is *complete* with respect to X, i.e. for every $w \in I_X$, if $\Theta_X(w) = 1$, then there exists a computation of $\Pi(w)$ which is an accepting computation.

We then say that a decision problem $X = (I_X, \Theta_X)$ is solvable in polynomial time by a family of $\mathbf{\Pi} = \{\Pi(w) | w \in I_X\}$ of SNPSP systems, in a nondeterministic and semi-uniform way if there exists a $k \in \mathbf{N}$ such that X is solvable by the family $\mathbf{\Pi}$ in time bounded by a polynomial, in a nondeterministic and semi-uniform way.

The **NP**-complete problem considered here, `Subset sum`, can be defined as follows:
Problem: `Subset Sum` [31]

- Instance: S, and a (multi)set $V = \{v_1, v_2, \ldots, v_n\}$, with $S, v_i \in \mathbb{N}$ and $1 \leq i \leq n$;
- Question: Is there a sub(multi)set $B \subseteq V$ such that $\sum_{b \in B} b = S$?

In [51], the `Subset sum` problem was also solved in a nondeterministic and semi-uniform way

using SNP systems with extended rules. Additionally, their solution used modules that have neurons operating in deterministic or nondeterministic ways, and applying rules in sequential or maximally parallel manner. A follow-up and improved (uniform) solution was then given in [52]. For the solutions we present here, unless otherwise mentioned, the SNPSP systems work in nonsaving mode as in Section 7.2. We note also that the solutions presented in this chapter are *nonconfluent*, i.e. there exist halting computations that are accepting while other computations are non-accepting, see e.g. [22][84]. There are several ways of encoding an instance of `Subset Sum` as the input to the system. Two common ways (used in this work, and as in [51] and [52]) involve either (`i`) starting with an initial configuration where each σ_i stores v_i number of spikes, $1 \leq i \leq n$, or (`ii`) each σ_{in_i} receives a number of spikes from the environment equal to non-zero multiples of $v_i, 1 \leq i \leq n$.

10.2 A semi-uniform solution to `Subset Sum`

We begin by providing a family $\mathbf{\Pi}$ of nondeterministic and semi-uniform SNPSP systems solving `Subset Sum` in constant time. The size of each $\Pi \in \mathbf{\Pi}$ is dependent on the value of n and each $v_i, 1 \leq i \leq n$. The input value v_i is already embedded in the system as initial number of spikes of the corresponding neurons.

Theorem 10.2.1. *The* `Subset Sum` *problem is solvable by a family* $\mathbf{\Pi}$ *of SNPSP systems in a nondeterministic and semi-uniform way, where each* $\Pi \in \mathbf{\Pi}$ *have the following restrictions (normal form)*

- *synapse level nondeterminism only,*
- *without forgetting rules and rules with delays,*
- *computes in constant time.*

Proof. First, we formally describe each SNPSP system $\Pi_{\mathtt{ss}}(W), \Pi_{\mathtt{ss}} \in \mathbf{\Pi}$, that solves instance W of the problem.

$\Pi_{\mathtt{ss}} = (\{a\}, \sigma_1, \ldots, \sigma_n, \sigma_{1^{(1)}}, \ldots, \sigma_{1^{(2v_1)}}, \ldots, \sigma_{n^{(1)}}, \ldots, \sigma_{n^{(2v_n)}}, \sigma_{out}, syn, out)$ where:

(**1**) For $1 \leq i \leq n, \sigma_i = (v_i, R_i)$, with $R_i = \{a^{v_i} \rightarrow +v_i(v_i, \{i^{(1)}, \ldots, i^{(2v_i)}\})\}$;

(**2**) For $1 \leq j \leq v_n, \sigma_{i^{(j)}} = (0, \{a \rightarrow a\})$;

(**3**) For $v_n + 1 \leq k \leq 2v_n, \sigma_{i^{(k)}} = (0, \emptyset)$;

and

- $\sigma_{out} = (0, \{a^S \rightarrow a\})$;
- $syn = \{(1^{(1)}, out), \ldots, (1^{(v_n)}, out), \ldots, (n^{(1)}, out), \ldots, (n^{(v_n)}, out)\}$;

We refer to Figure 10.1 for a graphical representation of $\Pi_{\mathtt{ss}}$. In Figure 10.1, for simple neurons that only have the rule $a \rightarrow a$, we omit the writing of the rule as in Section 7.4. Note however that in the definition of $\Pi_{\mathtt{ss}}$, some neurons actually do not have rules in them. The distinction

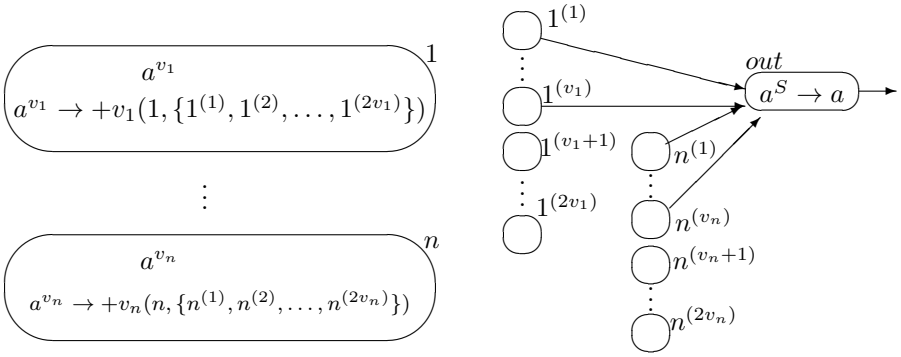

Figure 10.1: The semi-uniform SNPSP system $\Pi_{\texttt{ss}}$ solving `Subset Sum`.

between these empty neurons and the simple neurons will be explained shortly. Next, we provide the functioning of $\Pi_{\texttt{ss}}$, which we divide into three stages for clarity:

Stage 1: This stage consists of operations performed by neurons σ_1 to σ_n, where σ_i initially contains v_i spikes. These are the neurons from (**1**). Each σ_i consumes all the initial v_i spikes, to create v_i number of synapses. Notice that each σ_i has $\alpha = +$ and $k = v_i$, but can nondeterministically select among $2v_i$ number of neurons to create a synapse to. Once every σ_i has finished selecting and creating synapses to v_i number of neurons, we move to the next stage.

Stage 2: This stage consists of operations performed by neurons $\sigma_{1^{(1)}}$ up to $\sigma_{1^{(2v_1)}}$ associated with σ_1 (from stage 1), up to the neurons $\sigma_{n^{(1)}}$ up to $\sigma_{n^{(2v_n)}}$ associated with σ_n (from stage 1). These are the neurons from (**2**) and (**3**). Note in the definition that given the $2v_i$ neurons in this stage associated with σ_i in stage 1, the first half of these neurons (i.e. $\sigma_{i^{(1)}}$ up to $\sigma_{i^{(v_i)}}$) are simple neurons with the rule $a \to a$. The second half (i.e. $\sigma_{i^{(v_i+1)}}$ up to $\sigma_{i^{(2v_i)}}$) are empty neurons, without initial spikes or rules. Also note that neurons in the first half each have a synapse to σ_{out}, while the second half do not have synapses, i.e. they are all traps, where each of their presynaptic sets is empty.

Since the nondeterminism in stage 1 is synapse level, in some computations some (or all) simple neurons will each receive a spike, while in other computations some (or all) empty neurons each receive a spike. For the computations that send a spike to the empty neurons, these neurons function as trap or repository neurons. Consequently, the spikes in these trap neurons can also be removed using forgetting or plasticity rules. The computations that have some (or all) simple neurons in this stage then send one spike each to σ_{out}, leading us to the next stage.

Stage 3: This final stage checks the existence of an S number of spikes from stage 2. If there are S spikes in σ_{out} this means the answer to this instance of the `Subset Sum` is affirmative, and one spike is sent to the environment. This spike, affirming the answer to the problem instance, is sent out to the environment 3 time steps since stage 1. If however the spikes sent to σ_{out} from stage 2 do not add up to S, this means the answer to the problem instance is negative. Therefore, at time

step 3 no spike is sent to the environment. All Theorem parameters are clearly satisfied. □

In $\Pi_{\texttt{ss}}$ the computation time is constant (3 steps) but the number of neurons is dependent on the individual values of the input numbers $v_i, 1 \leq i \leq n$. At the price of slightly extending the computation time (it is still constant), we obtain a uniform solution in the following section.

10.3 A uniform solution to `Subset Sum`

Next, we provide a family $\mathbf{\Pi}$ solving `Subset Sum` in a nondeterministic and uniform way in constant time. In this case, the system only "knows" the value of n, while S and $v_i, 1 \leq i \leq n$ must be introduced into the system using $n+1$ input neurons.

Theorem 10.3.1. *The* `Subset Sum` *problem is solvable by a family* $\mathbf{\Pi}$ *of SNPSP systems in a nondeterministic and uniform way, where each* $\Pi \in \mathbf{\Pi}$ *have the following restrictions (normal form)*

- *synapse level nondeterminism only,*
- *without using delays,*
- *computes in constant time.*

Proof. Each SNPSP system $\Pi_{\texttt{us}}(W), \Pi_{\texttt{us}} \in \mathbf{\Pi}$, that solves instance W of the problem is illustrated in Figure 10.2. We introduce $2v_i, 1 \leq i \leq n$ spikes into the corresponding input neuron in_i, while we introduce $2S$ spikes into in_{n+1}. Figure 10.2 shows these spikes already present in the $n+1$ input neurons. These even number of spikes cannot be used by σ_{in_i} until a spike is received from σ_{c_i}. Neurons $\sigma_{c_i}, 1 \leq i \leq n$ are the only neurons with nondeterminism (synapse-level). These neurons nondeterministically allow their corresponding σ_{in_i} neurons to spike (if σ_{c_i} creates synapse (c_i, in_i)) or not (if σ_{c_i} creates synapse (c_i, x)). Note that there is no σ_x so that (c_i, x) is never actually created.

Neurons σ_{c_i} apply their rules at step 1, and at step 3 neurons $e_{i,1}$ spike if they become activated from their corresponding in_i neuron. It also takes 3 steps before σ_{h_3} and σ_{h_4} begin to spike, starting with the spiking of σ_{h_1} at step 1. Neurons σ_{h_3} and σ_{h_4} "feed" a spike to each other starting at step 3. A spike is also sent from σ_{h_4} to σ_{t_1} (the "comparison trigger" neuron). At step 4 the odd number of spikes in σ_{t_1}, sent by the activated $e_{i,1}$ neurons and σ_{h_4}), allows the use of its forgetting rule to remove its all the spikes.

At step 5 only σ_{h_4} sends a spike to σ_{t_1}. At step 6, σ_{t_1} sends one spike each to σ_{h_4} and σ_{t_2}, while σ_{t_1} receives one more spike from σ_{h_4}. At step 7, σ_{h_4} stores two spikes so it can never spike again, while σ_{t_1} sends one more spike to σ_{t_2}. At step 8, σ_{t_2} sends a spike to σ_{acc} and $\sigma_{n_{n+1}}$ which become activated with an odd number of spikes now. Both σ_{acc} and $\sigma_{n_{n+1}}$ empty their spikes (removing two spikes each step) while sending one spike each to σ_{out}.

If the number of spikes accumulated in σ_{acc} equals the $2S$ number of spikes in $\sigma_{n_{n+1}}$, then the system will halt without producing a spike to the environment. Otherwise, σ_{out} will receive one spike

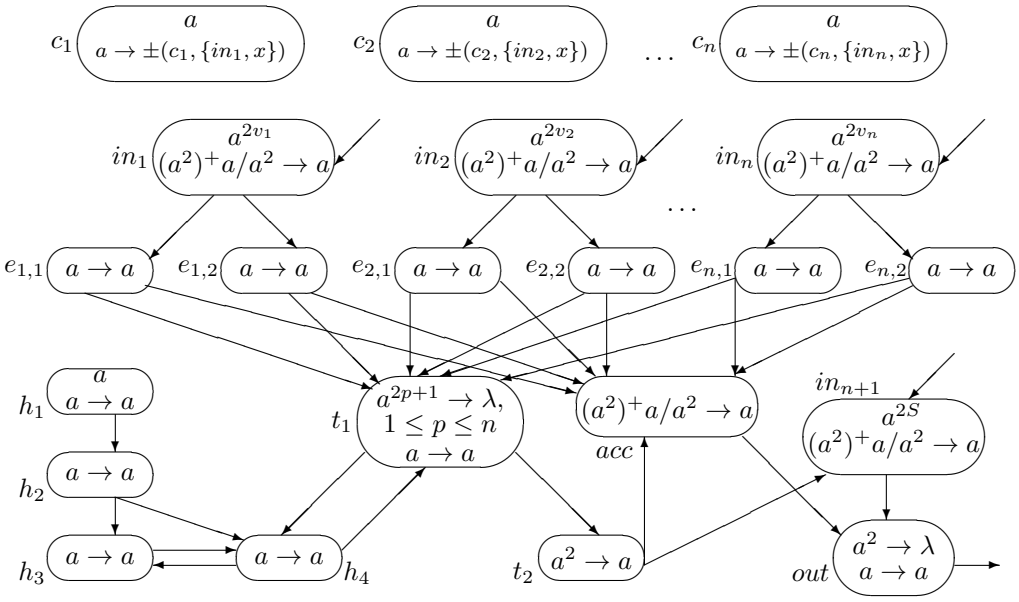

Figure 10.2: The uniform SNPSP system $\Pi_{\texttt{us}}$ solving `Subset Sum`.

from either σ_{acc} or $\sigma_{n_{n+1}}$ and send one spike to the environment. Halting without σ_{out} producing a spike, and σ_{out} producing a spike and halting at a later time, corresponds to an affirmative and negative answer to the problem instance, respectively.

The number of neurons is constant, with $4n+9$ neurons. The system halts in at most $3\sum_{i=1}^{n} v_i + 4$ steps: we have one initial step, at most $2max\{v_i | 1 \le i \le n\}$ to move the spikes from σ_{in_i} to σ_{t_1}, σ_{acc} and $\sigma_{in_{n+1}}$ become activated after four steps since σ_{t_1} became activated, then at most $\sum_{i=1}^{n} v_i$ steps for comparison. Since $max\{v_i | 1 \le i \le n\} \le \sum_{i=1}^{n} v_i$, we obtain the upper bound for the halting time. All Theorem parameters are satisfied. □

Actually, the forgetting rules in $\Pi_{\texttt{us}}$ can be removed by using a plasticity rule that functions like a forgetting rule, which is done by the σ_{c_i} neurons (synapse (c_i, x) is never created). The number of neurons and the halting time will still remain the same. In comparison, the non-uniform system in [51] solving `Subset Sum` (using forgetting rules, rules with delays, and standard rules) computes in four steps, while their uniform solution halts in at most $3\sum_{i=1}^{n} v_i + 6$ steps using $5n + 13$ neurons (also using delays, forgetting rules, and standard rules). Thus, one benefit of using synapse-level nondeterminism in this case is decreasing the needed neurons by a linear amount.

Part III

Final Remarks

Chapter 11

Conclusions

In this chapter, conclusions from the results in this dissertation are provided. Results from each chapter of Part II are briefly recalled, including conclusions from the results of each chapter. A general conclusion from the results in this dissertation is provided lastly.

Chapter 5 presented "routing simulations" using the routing modules for sequential, join, and split routing. The halting time of both Π (module with delay) and $\overline{\Pi}$ (module without delay) exactly coincide with one another (restriction R1), while the number of spikes in the environment for both systems are equal at halting time (restriction R2). These routing modules and simulations are meant to be used as sub-modules for building larger SNP systems, whether for number or language generating (accepting) purposes, guaranteed by R1 and R2. As such, we preserve spikes so that no spike is lost during the route simulation between the two systems. The trade off of the routing simulation however is that there is an "explosion" of neurons in $\overline{\Pi}$ for every delay in Π, i.e. we add d_i neurons in $\overline{\Pi}$ for every σ_i that has a delay. This explosion is expected, which can also be seen in constructions given in [51] for example, where they use either delays (with fewer number of neurons) or without delays (larger number of neurons) for nondeterministic generation of variables. These results are used to provide the main result in the chapter given in Theorem 5.2.1 regarding routing simulations in simple, semi-homogeneous SNP systems.

In Chapter 6, SNP modules were further investigated: amendments were made to the constructions for simulating DFA and DFST while reducing the amount of neurons for each module (given by Theorem 6.2.3 and Theorem 6.2.4). The single neuron in an SNP module is enough to simulate a DFA or a DFST, where a rule in the neuron simulates a transition in the simulated finite automata, i.e. given a simulated DFA or DFST with m number of transitions, then the simulating SNP module with neuron i has $|R_i| = m$. k-DFAO were also simulated using SNP modules containing two neurons (Theorem 6.3.1). The general idea is that the first neuron simulates the transition function while the second neuron simulates the output function of the k-DFAO. Using the SNP module simulations, we were able to generate automatic sequences with SNP modules (Theorem 6.3.2), as well as transfer some robustness properties of k-DFAO to the simulating SNP modules (Theorem 6.3.3). The constructions in Chapter 6 are also optimal, in the sense that they are the smallest modules in

terms of neuron count (one neuron in a module) that can be used to simulate the finite automata that were investigated. The constructions also have some similarities with the construction idea of [61], where a boundary for universality and non-universality in SNP systems with extended rules is 4 and 3 neurons, respectively.

Chapter 7 introduced SNPSP systems, which is the general focus of this dissertation. We introduced the structural plasticity feature in the SNP systems framework, providing a partial answer to an open problem in [80] about dynamism only for synapses. The "programming" of the system is dependent on the way neurons (dis)connect to each other using synapses. We proved that SNPSP systems as number generating and accepting devices are universal, for both the saving and nonsaving modes. The universality results hold even if we impose the following normal form on SNPSP systems: only 5 among the 17 total neurons for all modules[1] use plasticity rules, only synapse level nondeterminism exist, neurons without plasticity rules are simple (having only the rule $a \to a$), and without using forgetting rules or delays. Additionally, a state known as deadlock can arise during saving mode. Reaching such a state for an arbitrary SNPSP system in saving mode is undecidable.

The universality proofs of SNPSP systems are "easier" in the sense that they require less neurons in each module, while being under a normal form. The reason for this is the use of plasticity rules and their accompanying semantics: SNPSP systems achieve universality by creating or deleting at most k synapses each step, where $k \geq 1$. Interestingly, a seemingly minor change in the semantics of plasticity rule application allows SNPSP systems to maintain universality, although a deadlock can occur. Plasticity rules allow a change in the "flow" of spikes, by creating or deleting synapses, thereby redirecting the computation using fewer number (compared to "standard" SNP systems as in [44]) of rules or neurons. The results of succeeding chapters further elaborated the benefits of plasticity rules.

From Chapter 8, we showed that SNPSP systems (with induced sequentiality) as acceptors are more powerful than as generators, since the former do not need any source of nondeterminism. SNPSP systems maintain Turing universality despite the following restrictions: induced sequentiality, and under a normal form. The normal form is that purely plastic neurons have at most two rules (the maximum in the system), and neurons with standard rules are simple. Acceptors are universal in all four modes, while generators are not universal under *maxps* and *minps* modes. Replacing the commonly used nd_{rule} (e.g. in [44] and [41]) with nd_{syn} as a nondeterminism source, we are able to: allow generators in *maxps* or *minps* mode to become universal; reduce the rules in each neuron to at most two, while making the modules more compact, i.e. require fewer neurons.

Furthermore, our results provide a family of uniform modules, i.e. at the expense of using nd_{syn}, our modules can be used in either *maxs* and *maxps*, or *mins* and *minps*. In [47], the problem of constructing a family of uniform ADD, SUB, and FIN modules that can be used in both *mins* and *minps* modes was open. Results in this chapter thus provide a positive hint to this open problem.

[1] ADD, SUB, and FIN module neurons in the saving and generative case, since the accepting case requires a lesser number of neurons with plasticity rules.

$NRE = N_2SNPSP.$	Theorem 7.4.1
$NRE = N_{acc}SNPSP.$	Theorem 7.4.2
$NRE = N_2SNPSP_s.$	Theorem 7.5.1
$NRE = N_{2,gen}SNPSP^{maxs}(rule_2, \pm syn_k, nd_{syn}), k \geq 1.$	Theorem 8.3.1
$NRE = N_{2,gen}SNPSP^{maxs}(rule_2, \pm syn_k), k \geq 2.$	Theorem 8.3.2
$NRE = N_{acc}SNPSP^{maxs}(rule_2, \pm syn_k), k \geq 1.$	Theorem 8.3.3
$NRE = N_{2,gen}SNPSP^{maxps}(rule_2, \pm syn_k, nd_{syn}), k \geq 1.$	Theorem 8.3.4
$NRE = D_{acc}SNPSP^{maxps}(rule_2, \pm syn_k), k \geq 1.$	Theorem 8.3.5
$NRE = N_{2,gen}SNPSP^{mins}(rule_2, \pm syn_k), k \geq 2.$	Theorem 8.3.6
$NRE = N_{acc}SNPSP^{mins}(rule_2, \pm syn_k), k \geq 2.$	Theorem 8.3.7
$NRE = N_{2,gen}SNPSP^{minps}(rule_2, \pm syn_k, nd_{syn}), k \geq 2.$	Theorem 8.3.8
$NRE = N_{acc}SNPSP^{minps}(rule_2, \pm syn_k), k \geq 2.$	Theorem 8.3.9
$SLIN = N_{tot}SNPSP^{asyn}(bound_p, nd_{syn}), p \geq 1.$	Theorem 9.3.1
$NRE = N_{tot}WSNPSP^{asyn}(rule_m, \pm syn_k, weight_l, nd_{syn}), m \geq 9, k \geq 1, l \geq 3.$	Theorem 9.3.2

Table 11.1: Summary of main (non)universality results concerning SNPSP systems in this work.

From Chapter 9, we investigated the computability of asynchronous SNPSP systems. In [19] it is known that asynchronous SNP systems with extended rules are universal, while the conjecture is that asynchronous SNP systems with standard rules are not [18]. In Theorem 9.3.1, we showed that asynchronous bounded SNPSP systems are not universal where, similar to standard rules, each neuron can only produce at most one spike each step. In Theorem 9.3.2, asynchronous WSNPSP systems are shown to be universal. In WSNPSP systems, the synapse weights perform a function similar to extended rules in the sense that a neuron can produce more than one spike each step. Our results thus provide support to the conjecture about the nonuniversality of asynchronous SNP systems with standard rules.

From Chapter 10, we provided two families of solutions to the **NP**-complete problem `Subset Sum`. The semi-uniform family of solution halts in a fewer number of steps (Theorem 10.2.1), compared to the uniform family of solutions (Theorem 10.3.1), although both halt in a constant number of steps. Both families do not use rules with delays, and only use nd_{syn} as nondeterminism source. The uniform family uses forgetting rules (unlike the semi-uniform solution) but this technical detail can be removed by using plasticity rules in a way where plasticity rules function as forgetting rules. The use of plasticity rules also decreases the number of neurons in the uniform family by a linear amount, compared to the uniform family for `Subset Sum` given in [51]. This linear decrease in the number of neurons is due to the synapse-level nondeterminism, removing the need for extra neurons with rule-level nondeterminism in [51].

The contributions of this dissertation fall under the computability of SNP systems, starting with the simulation results provided in Chapter 5 and 6, followed by the introduction of and investigations on SNPSP systems. The neuroscience phenomenon of structural plasticity, taken from biological neurons, has provided inspiration resulting in SNPSP systems. A summary of the main (non)universality results in this work regarding SNPSP systems is provided in Table 11.1.

The "programming capacity" offered by plasticity rules in SNPSP systems can provide the "release" of common SNP systems features (delays in rules, forgetting rules) while having simplifying sets of restrictions (normal forms). The plasticity feature therefore is a useful and interesting addition to the SNP systems framework, where the feature is a response to the challenge **D** presented in [80], regarding SNP systems with dynamism applied only to synapses. Plasticity rules can also help reduce the number of neurons in the system through the use of synapse-level nondeterminism, for example in cases where nondeterministic selection or assignment of variables are necessary, as in Chapter 10 and in some universality proofs in previous chapters. While the plasticity feature has been identified in this work (mainly along the lines of computability theory) as interesting, much still remains to be investigated, as detailed and emphasized by the succeeding and final chapter.

Chapter 12

Further work

In this chapter, we end the dissertation by providing some future research directions which are directly or indirectly of interest to the results in this work.

From Chapter 5 results, it is interesting to realize constructions for non-simple or non-homogeneous SNP systems. Minimization of the number of neurons (likely, with a trade-off) of a $\overline{\Pi}$ route simulating a Π is also desirable, including providing bounds, e.g. the number of neurons, spikes.

From Chapter 6, we mention that in [21], strict inclusions for the types of languages characterized by SNP systems with extended rules having one, two, and three neurons were given. Then in [77], it was shown that there is no SNP transducer that can compute nonerasing and nonlength preserving morphisms: for all $a \in \Sigma$, the former is a morphism h such that $h(a) \neq \lambda$, while the latter is a morphism h where $|h(a)| \geq 2$. It is known (e.g. in [3]) that the Thue-Morse morphism is given by $\mu(0) = 01$ and $\mu(1) = 10$. It is interesting to further investigate SNP modules with respect to other classes of sequences, morphisms, and finite transition systems. Another technical note is that in [77] a time step without a spike entering or leaving the system was considered as a symbol of the alphabet, while in [43] (and in this work) it was considered as λ.

We also leave as an open problem a more systematic analysis of input/output encoding size and system complexity: in the constructions for Theorems 6.2.3 to 6.2.4, SNP modules consist of only one neuron for each module, compared to three neurons in the constructions of [43]. However, the encoding used in our theorems is more involved, i.e. with multiplication and addition of indices (instead of simply addition of indices in [43]). On the practical side, SNP modules might also be used for computing functions, as well as other tasks involving streams of (multiple) input-output transformations. Practical applications might include image modification or recognition, sequence analyses, online algorithms, et al. Perhaps improving or extending the work done in [24].

Some preliminary work on SNP modules and morphisms was given in [9]. From finite sequences, it is interesting to extend SNP modules to infinite sequences. In [27], extended SNP systems were used as acceptors in relation to ω-languages. SNP modules could also be a way to "go beyond Turing" by way of *interactive computations*, as in interactive components or transducers given in [32]. While the syntax of SNP modules may prove sufficient for these "interactive tasks", a change

in the semantics is probably necessary.

With the introduction of SNPSP systems in Chapter 7, we can have the following: The deadlock state does not seem to exist in nonsaving mode. Does this mean that saving mode is "better" than nonsaving mode? Both modes have the same expressiveness, but perhaps the (non)existence of a deadlock is useful for more practical purposes, e.g. modeling or analysis of systems. For example, reaching a deadlock can be interpreted as reaching an unwanted state or fault in a network or system. This is an interesting theoretical and practical question. Deadlocks and other state types, as applied to modeling and analysis, have an extensive body of research in many graphical formalisms such as Petri nets [60][85]. Several works have related Petri nets, SNP systems, and other P systems. See for example [11] and references therein. Deadlock can also occur in systems in [76], inspiring the undecidability result in the chapter.

Other parameter modifications of computational interest abound: what about parallel synapse creation-deletion? The semantics in Section 7.2 for $\alpha \in \{\pm, \mp\}$ is sequential, requiring two time steps to perform such rules with such values for α. We only list here a few other complexity parameters that can be used: the size of *syn* during a computation, e.g. the number of synapses created (deleted), or a bound on the size of *syn* (related to synaptic homeostasis); in the universality proofs we had $\alpha \in \{+, -, \pm\}$, so can we have universality with α having at most two values only? Normal forms for "standard" SNPSP systems are interesting as well, as provided in this section are interesting. For example in [71], it is proven that two values for α suffice for universality.

SNPSP systems and the proofs in Section 7.4 are reminiscent of the more generalized extended SNP systems[1] in [2] which, due to their generality, have produced impressive computability results but lacks biological motivations.[2] Investigations of ESNP and SNPSP systems further seems interesting, especially since the former are generalized versions of SNP systems while the latter are more restricted but with bio-inspirations. More recently, SNP systems with rules on synapses were shown to be universal [91]. In these systems, neurons are only spike repositories, and the spike processing (including nondeterminism) are done in the synapses. Investigating theoretical or practical usefulness of such systems, together with structural plasticty, is also interesting. Perhaps Hebbian SNP systems in [33] and as mentioned in Section 7.1 could be combined with SNPSP systems, thereby providing a framework which includes both functional and structural plasticity. This combined framework could perhaps be further investigated along the lines of, among others, machine learning.

SNPSP systems equipped with input and one output neurons can be used as transducers, to compute (in)finite strings as in [77]. Furthermore, SNPSP systems with multiple inputs or outputs are also interesting, for computing strings or vectors of numbers as in [2] again.

From the sequential SNPSP systems in Chapter 8, it is also interesting to consider the notion of homogeneous systems. In [48] for example, two types of neurons are enough for universality:

[1] Not to be confused with SNP systems using (or with) extended rules.

[2] Personal communications with Rudolf Freund, 13th Brainstorming Week in Membrane Computing, Sevilla, Spain, 2 to 6 February 2015.

one for *maxs*, and another for *maxps*. It is then proved that homogeneous SNP (HSNP) systems as acceptors and generators are universal in both modes. Since neurons are homogeneous, the connectivity of the neurons is important. For SNPSP systems, perhaps a pseudo-homogeneous construction is reasonable, where only the set N is different among rules of neurons.

In asynchronous SNPSP systems from Chapter 9, it is interesting to realize the computing power of asynchronous unbounded (in spikes) SNPSP systems, since we know that bounded ones are not universal. Again, checking the regular expressions in these systems is interesting. It can be argued that when $\alpha \in \{\pm, \mp\}$, the synapse creation (resp., deletion) immediately followed by a synapse deletion (resp., creation) is another form of synchronization. Can asynchronous WSNPSP systems maintain their computing power, if we further restrict them by removing such semantic? Another interesting question is as follows: in the ADD module in Theorem 9.3.2, we have nd_{syn}. Can we still maintain universality if we remove this level, so that nd_{neur} in *asyn* mode is the only source of nondeterminism? In [19] for example, the modules used *asyn* mode and nd_{rule}, while in [89], only *asyn* mode was used (but with the use of a new ingredient called local synchronization).

In Theorem 9.3.2, the construction is based on the value $|S_r|$. Can we have a uniform construction while maintaining universality? i.e. can we construct a Π such that $N(\Pi) = NRE$, but is independent on the number of `SUB` instructions of M? Then perhaps parameters m and l in Theorem 9.3.2 can be reduced.

From Chapter 10, it is interesting to ask other ways to encode instances of other **NP**-complete problems for (non)deterministic synapse selection. In computer science we work to have efficient solutions, seeking minimal parameters. However, and as we mentioned Chapter 4, biology is usually not space efficient: there are billions of neurons, each with thousands of synapses in our brains. So the semi-uniform solution may not immediately appeal to our search for minimal parameters, but it seems biologically appealing. Depending on the instance to be solved, the number of neurons for example (and possibly the spikes and synapses) in $\Pi_{\texttt{ss}}$ can be exponential with respect to the instance size. The uniform solution can use much fewer neurons than the semi-uniform solution, but again, both presented solutions are nondeterministic. How can we use plasticity rules to provide deterministic solutions? Again, adding the constraint that our solution must be deterministic, it is likely that we must pay the price of "space" i.e. exponential number of neurons in the system. Confluent solutions are also desirable, especially with the interest of simulating or implementing solutions with traditional hardware or software.

From the above mentioned problems directly related to the results in this work, much is left to be investigated for SNPSP systems. We here only list a few established results in membrane computing and SNP systems which can be applied to SNPSP systems: investigating the type of regular expression in (e.g. sequential, asynchronous) standard or plasticity rules used, e.g. bounded, unbounded, or general rules as in [19] and [89]; constructing small universal systems, another active line of investigation in models of computation, as in [67], [78] and [103]; this dissertation focused

on the generating or accepting of numbers, but another well investigated area in SNP systems and its many variants is generating or accepting languages as in [21][27][42]; the processing of (in)finite sequences, similar to SNP transducers as in [75]; the exhaustive use of rules (maximal parallelism) as in [68].

We also emphasize that most results in this dissertation on SNPSP systems, regardless of semantics or mode, operated under in nonsaving mode. Further investigation on this semantic for SNPSP systems, among other semantics, seems interesting to apply to the open problems mentioned above. The idea of using semantics rather than syntax, as another frontier for tractability, has recently been raised in the recently concluded Brainstorming Week on Membrane Computing 2015.[3] It is also interesting to investigate the possibility of hierarchies in SNP and SNPSP systems, whether universal or not, depending on parameters mentioned here or otherwise. For example the techniques in [38] could be used for other P systems other than those considered in the article (restricted communicating P systems). Creation of a synapse is analogous to opening a "communication channel" between two neurons, while deleting synapses is analogous to removing a channel. In several of the proofs presented here using SNPSP systems, in certain cases plasticity rules can operate similar to a forgetting rule or a rule with delay. An interesting line of investigation is to provide a more systematic analysis and simulation (perhaps structurally or behaviourally) between plasticity rules, spiking rules (with or without delays), and forgetting rules (among other features).

SNPSP systems could also be investigated together with arithmetic operations as in [102]. Recently, SNPSP systems were used to simulate Boolean circuits in [86]. Since the connectivity of neurons in SNPSP systems is important, a more systematic study of the (non)existence and effects (e.g. on the computing power, applications) of cycles or loops in the system graph also seems promising theoretically and practically. Perhaps SNPSP systems, with certain modifications in syntax and semantics, could be applied to networks (e.g. traffic) where switching between several routes or options is common. The switching could be considered as a (non)deterministic selection of which synapse (or route, path etc.) to connect or disconnect. Further investigations also on SNPSP systems can also provide ideas on their linear algebra properties. The graphics processing unit simulators in [7] and [8] for example, are based on the work in [101]. In [101], SNP systems without delays, their configurations, and computations, were represented as matrices, vectors, and linear algebra operations.

We end this dissertation with the firm belief, first mentioned in Chapter 1, that much inspiration from nature still remains, with the goal of moving the human condition forward. Quoting the final line in "Computing Machinery and Intelligence" by A.M. Turing (published in *Mind*, 59, pp. 433-460, 1950): *"We can only see a short distance ahead, but we can see plenty there that needs to be done."*

[3] See also the related presentation in `http://www.gcn.us.es/files/bwmc2015_gutierrez.pdf`.

List of References

[1] Adleman, L.M. (1994) Molecular computation of solutions to combinatorial problems. Science vol. 266(5187), pp. 1021-1024.

[2] Alhazov, A., Freund, R., Oswald, M., Slavkovik, M. (2006) Extended Spiking Neural P Systems Generating Strings and Vectors of Non-Negative Integers. In: Hoogeboom, HJ et al. (eds.) WMC7, LNCS vol. 4361, pp. 123-134.

[3] Allouche, J-P., Shallit, J. (2003) Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press.

[4] Bray, D. (1990) Intracellular signalling as a parallel distributed process. J. Theoretical Biology vol 143, pp. 215-231.

[5] Butz, M., Wörgötter, F., van Ooyen, A. (2009) Activity-dependent structural plasticity. Brain Research Reviews vol. 60, pp. 287-305.

[6] Binder, A., Freund, R., Oswald, M., Vock, L. (2007) Extended spiking neural P systems with excitatory and inhibitory astrocytes. In: Gutiérrez-Naranjo, M. A. et al (eds.), Proc. 5th Brainstorming Week on Membrane Computing pp. 63-72, Sevilla, Fenix Editora.

[7] Cabarle, F.G.C., Adorna, H.N., Martínez-del-Amor, M.A. (2012) A Spiking Neural P system simulator based on CUDA. In: Gheorghe, M. et al. (eds.), CMC12, LNCS vol. 7184, pp. 87-103.

[8] Cabarle, F.G.C., Adorna, H.N., Martínez-del-Amor, M.A., Pérez-Jiménez, M.J. (2012) Improving GPU Simulations of Spiking Neural P Systems. Romanian Journal of Information Science and Technology. vol. 15(1), pp. 5-20.

[9] Cabarle, F.G.C., Buño, K.C., Adorna, H.N. (2012) Spiking Neural P Systems Generating the Thue-Morse Sequence. Asian Conference on Membrane Computing 2012, Wuhan, China, 15-18 October.

[10] Cabarle, F.G.C., Buño, K.C., Adorna, H.N. (2012) On The Delays in Spiking Neural P Systems. Philippine Computing Journal, vol. 7(2), pp. 12-17.

[11] Cabarle, F.G.C., Adorna, H.N. (2013) On Structures and Behaviors of Spiking Neural P Systems and Petri Nets. In: Csuhaj-Varju, E. et al. (eds.): CMC13, LNCS vol. 7762, pp.145-160 .

[12] Cabarle, F.G.C., Adorna, H.N., Ibo, N. (2013) Spiking neural P systems with structural plasticity. 2nd Asian Conference on Membrane Computing. Chengdu, China, pp. 13 - 26, 4-7 November.

[13] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J., Song, T. (2015) Spiking neural P systems with structural plasticity. (to appear) Neural Computing and Applications doi:10.1007/s00521-015-1857-4.

[14] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Notes on Spiking Neural P Systems and Finite Automata. (to appear) 13th Brainstorming Week on Membrane Computing, Sevilla, Spain.

[15] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Asynchronous spiking neural P systems with structural plasticity. (to appear) 14th Unconventional Computation and Natural Computation, Auckland, New Zealand, and In: Calude, C., Dinneen, M. (eds.) LNCS vol. 9252 doi:10.1007/978-3-319-21819-9.

[16] Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J. (2015) Sequential spiking neural P systems with structural plasticity based on max/min spike number. (to appear) Neural Computing and Applications doi:10.1007/s00521-015-1937-5.

[17] Cardona, M. Angels Colomer, M., Margalida, A., Palau, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D. (2010) A computational modeling for real ecosystems based on P systems. Natural computing vol. 10(1), pp. 39-53.

[18] Cavaliere, M., Egecioglu, O., Woodworth, S., Ionescu, I., Păun, G. (2008) Asynchronous spiking neural P systems: Decidability and undecidability. DNA 2008, LNCS vol. 4848, pp. 246-255.

[19] Cavaliere, M., Ibarra, O., Păun, G., Egecioglu, O., Ionescu, M., Woodworth, S. (2009) Asynchronous spiking neural P systems. Theoretical Computer Science, vol. 410, pp. 2352-2364.

[20] Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A. Pérez-Hurtado, I., Pérez-Jiménez, M.J. (2010) Simulation of P systems with active membranes on CUDA. Briefings in Bioinformatics vol. 11(3), pp. 313-322.

[21] Chen, H., Ionescu, M., Ishdorj, T-O., Păun, A., Păun, G., Pérez-Jiménez, M.J. (2008) Spiking neural P systems with extended rules: universality and languages. Natural Computing, vol. 7, pp. 147-166.

[22] Ciobanu, G., Păun, G., Pérez-Jimeńez, M.J. (2006) Applications of Membrane Computing, Springer-Verlag Berlin Heidelberg.

[23] Cooper, S.B., Löwe, B., Sorbi, A. (eds.) (2008) New Computational Paradigms: Changing Conceptions of What is Computable. Springer-Verlag New York.

[24] Díaz-Pernil, D., Peña-Cantillana, F., Gutiérrez-Naranjo, M.A. (2013) A parallel algorithm for skeletonizing images by using spiking neural P systems. Neurocomputing, vol. 115, pp. 81-91.

[25] Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A. (2006) Available membrane computing software. In [22] pp. 411-436.

[26] Díaz-Pernil, D. Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A. (2009) A P-lingua programming environment for Membrane Computing. LNCS vol. 5391, pp. 187-203.

[27] Freund, R., Oswald, M. (2008) Regular ω-languages defined by finite extended spiking neural P systems. Fundamenta Informaticae, vol. 81(1-2), pp. 65-73.

[28] Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.) (2014) Applications of Membrane Computing in Systems and Synthetic Biology. Springer.

[29] García-Arnau, M., Pérez, D., Rodríguez-Patón, A., Sosík, P. (2008) On the power of elementary features in spiking neural P systems. Natural Computing vol. 7, pp. 471-483.

[30] García-Arnau, M., Pérez, D., Rodríguez-Patón, A., Sosík, P. (2009) Spiking Neural P Systems: Stronger Normal Forms. J. of unconventional computing, vol. 5(5), 411-425.

[31] Garey, M.R., Johnson, D.S. (1979) Computers and intractability. A guide to the theory on **NP**-Completeness. W.H. Freeman and Company, CA, USA.

[32] Goldin, D., Smolka, S., Wegner, P. (eds.) (2006) Interactive Computation: The New Paradigm. Springer-Verlag Berlin Heidelberg.

[33] Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J. (2009) Hebbian Learning from Spiking Neural P Systems View. In: Corne, D. et al. (eds.) WMC9, LNCS vol. 5391, 217-230.

[34] Gerstner, W., Kistler, W . (2002) Spiking Neuron Models: Single Neurons, Populations, Plasticity. Cambridge University Press.

[35] Ghosh-Dastidar, S., Adeli, H. (2009) Third Generation Neural Networks: Spiking Neural Networks. In: Yu, W. Sanchez, E.N. (eds) Advances in Computational Intelligence, AISC vol. 116, pp. 167-178.

[36] Gramss, T. (1998) The Theory of Quantum Computation: An Introduction. In: Gramss, T. (eds.) Non-Standard Computation: Molecular Computation - Cellular Automata - Evolutionary Algorithms - Quantum Computers. pp. 141-174, Wiley-VCH, Weinheim.

[37] Hopcroft, J., Motwani, R. , Ullman, J. (2001) Introduction to Automata Theory, Languages, and Computation. 2nd ed. Addison-Wesley.

[38] Ibarra, O. (2005) On membrane hierarchy in P systems. Theoretical Computer Science vol. 334(1-3), pp. 115-129.

[39] Ibarra, O., Woodworth, S. (2007) Spiking neural P systems: some characterizations. FCT 2007, LNCS vol. 4639, pp. 23-37.

[40] Ibarra, O., Păun, A., Păun, G., Rodríguez-Patón, A., Sosík, P., Woodworth, S. (2007) Normal forms for spiking neural P systems. Theoretical Computer Science vol. 372(2-3), pp. 196-217.

[41] Ibarra, O., Păun, A., Rodríguez-Patón, A.(2009) Sequential SNP systems based on min/max spike number. Theor. Com. Sci., vol. 410, pp. 2982-2991.

[42] Ibarra, O., Leporati, A., Păun, A., Woodworth, S. (2010) Chapter 13: Spiking neural P systems. In [81], pp. 337-362.

[43] Ibarra, O. , Peréz-Jiménez, M.J. , Yokomori, T. (2010) On spiking neural P systems. Natural Computing. vol. 9, pp. 475-491.

[44] Ionescu, M., Păun, G. , Yokomori, T. (2006) Spiking Neural P Systems. Fundamenta Informaticae, vol. 71(2,3), pp. 279-308.

[45] Ionescu, M., Sburlan, D. (2008) Some applications of spiking neural P systems. J. of Computing and Informatics, vol. 27(3), pp. 515-528.

[46] Ishdorj, T.-O., Leporati, A., Pan, L., Zeng, X.,Zhang, X. (2010) Deterministic solutions to QSAT and Q3SAT by spiking neural P systems with pre-computed resources. Theoretical Computer Science vol. 411, pp. 2345-2358.

[47] Jiang, K., Song, T., Pan, L. (2013) Universality of sequential spiking neural P systems based on minimum spike number. Theor. Com. Sci., vol. 499, pp. 88-97.

[48] Jiang, K., Song, T., Chen, W., Pan, L. (2013) Homogeneous spiking neural P systems working in sequential mode induced by maximum spike number. J. of Computer Mathematics, vol. 90(4), pp. 831- 844.

[49] Kari, L., Rozenberg, G. (2008) The Many Facets of Natural Computing. Communications of the ACM. vol. 51(10), pp.72-83.

[50] Kleene, S.C. (1956) Representation of Events in Nerve Nets and Finite Automata. Automata Studies. Princeton University Press, Princeton, NJ, pp. 3-42.

[51] Leporati, A., Zandron, C., Ferretti, C., Mauri, G. (2007) Solving Numerical NP-Complete Problems with Spiking Neural P Systems. In: Eleftherakis, G. et al. (eds.) WMC8 2007, LNCS vol. 4860, pp. 336-352.

[52] Leporati, G., Mauri, G., Zandron, C., Păun, G., Pérez-Jiménez, M. (2009) Uniform solutions to SAT and Subset Sum by spiking neural P systems. Natural Computing vol. 8, 681-702.

[53] Maass, W., Bishop, C. (eds.) (1999) Pulsed Neural Networks. MIT Press.

[54] Maass, W. (2002) Computing with spikes. Special Issue on Foundations of Information Processing of TELEMATIK, vol. 8(1) pp. 32-36.

[55] Macías-Ramos, F., Pérez-Hurtado, I., García-Quismondo, M., Valencia-Cabrera, L., Péerez-Jiménez, M.J., Riscos-Núñez, A. (2012) A P-Lingua based Simulator for Spiking Neural P Systems. In: Gheorghe, M. et al. (eds.) CMC12 LNCS vol. 7184, pp. 257-281.

[56] McCulloch, W.S. , Pitts, W.H. (1943) A Logical Calculus of the Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics, vol. 5, pp. 115-133.

[57] Martínez-del-Amor, M.A., García-Quismondo, M., Macías-Ramos, L.F., Valencia-Cabrera, L., Riscos-Nú uñez, A., Pérez-Jiménez, M.J. (2015) Simulating P Systems on GPU Devices: A Survey. Fundamenta Informaticae. vol. 136(3), pp. 269-284.

[58] Minsky, M. (1967) Computation: Finite and infinite machines. Englewood Cliffs, NJ: Prentice Hall.

[59] Minsky, M. (1961) Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. Ann. of Math., vol. 74, pp. 437- 455.

[60] Murata, T. (1989) Petri Nets: Properties, analysis and application. Proc. of the IEEE, vol. 77(4), pp. 541-580.

[61] Neary, T. (2010) A Boundary between Universality and Non-universality in Extended Spiking Neural P Systems. LATA2010, LNCS vol. 6031, pp. 475-487.

[62] Neary, T. (2010) On the computational complexity of spiking neural P systems. Natural Computing, vol. 9, pp. 831-851.

[63] Nishida, T.Y. (2004) An approximate algorithm for NP-complete optimization problems expoiting P systems. In: Proc. of the Workshop on Uncertainty in Membrane Computing. Palma de Mallorca, pp. 185-192.

[64] Nishida, T.Y. (2004) Membrane Algorithms: Approximate Algorithms for **NP**-Complete Optimization Problems. In [22], part 3, pp. 303-314.

[65] Pan, L., Păun, G. (2009) Spiking neural P systems with anti-spikes. J. of Computers, Communication, & Control. vol. IV(3), pp. 273-282.

[66] Pan, L., Păun, G. (2010) Spiking Neural P Systems: An improved normal form. Theoretical Computer Science vol. 411(6), pp. 906-918.

[67] Pan, L., Zeng, X. (2010) A Note on Small Universal Spiking Neural P Systems. WMC9 LNCS vol. 5957, pp.436-447.

[68] Pan, L., Zeng, X. (2011) Small Universal Spiking Neural P Systems Working in Exhaustive Mode. IEEE Trans. NanoBiosci., vol. 10(2), pp. 99-105.

[69] Pan, L., Păun, G., Pérez-Jiménez, M.J. (2011) Spiking neural P systems with neuron division and budding. Science China Information Sciences. vol. 54(8) pp. 1596-1607.

[70] Pan, L., Wang, J., Hoogeboom, J.H. (2012) Spiking Neural P Systems with Astrocytes. Neural Computation vol. 24, pp. 805-825.

[71] Pan, L., Song, T. (2015) A Normal Form of Spiking Neural P Systems with Structural Plasticity. (forthcoming) J. of Swarm Intelligence.

[72] Păun, G. (1998) Computing with membranes. Technical Report no. 208, Turku Center for Computer Science-TUCS, and (2000) Journal of Computer and System Science, vol. 61(1), pp. 108-143

[73] Păun, G. (2002) Membrane Computing: An Introduction. Springer-Verlag Berlin Heidelberg.

[74] Păun, G., Pérez-Jiménez, M.J. (2006) Membrane computing: Brief introduction, recent results and applications. Biosystems, vol. 85 pp. 11-22.

[75] Păun, G., Pérez-Jiménez, M.J., Rozenberg, G. (2006) Spike trains in spiking neural P systems. J. of Foundations of Computer Science, vol. 17(4), pp. 975-1002.

[76] Păun, G. (2007) Spiking Neural P Systems with Astrocyte-like Control. J. Universal Computer Science. vol. 13(11), pp. 1707-1721.

[77] Păun, G., Pérez-Jiménez, M.J., Rozenberg, G. (2007) Computing Morphisms by Spiking Neural P Systems. J. of Foundations of Computer Science. vol. 8(6) pp. 1371-1382.

[78] Păun, A., Păun, G. (2007) Small universal spiking neural P systems. BioSystems, vol. 90(1), pp. 48-60.

[79] Păun, G. (2008) From Cells to (Silicon) Computers, and Back. In [23] pp. 343-371.

[80] Păun, G., Pérez-Jiménez, M.J. (2009) Spiking Neural P Systems. Recent Results, Research Topics. In: Condon, A. et al. (eds.) Algorithmic Bioprocesses. pp. 273-291, Springer Berlin Heidelberg.

[81] Păun, G., Rozenberg, G., Salomaa, A. (eds.) (2010) The Oxford Handbook of Membrane Computing, Oxford University Press.

[82] Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Colomer, A. M., Riscos-Núñez, A. (2010) MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems. 5th IEEE BIC-TA, pp. 637-643.

[83] Pérez-Jiménez, M.J. (2005) An approach to computational complexity in membrane computing. In: Mauri, G. et al (eds.). WMC5 LNCS vol. 3365, pp. 85-109.

[84] Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F. (2006) Computationally Hard Problems Addressed Through P Systems. In [22], pp. 315-346.

[85] Reisig, W., Rozenberg, G. (eds) (1998) Lectures on Petri Nets I: Basic Models, and Lectures on Petri Nets II: Applications. Springer-Verlag Berlin Heidelberg.

[86] Rosini, B., Dersanambika, K.S. (2014) Simulation of Boolean Circuits using Spiking Neural P Systems with Structural Plasticity. J. of Computer Trends and Technology. vol. 11(1), pp. 1-9.

[87] Rozenberg, G. (2008) Computer Science, Informatics, and Natural Computing - Personal Reflections. In [23] pp. 373-379.

[88] Rozenberg, G., Bäck, T., Kok, J.N. (eds.) (2012) Handbook of Natural Computing. vol. 1-4, Springer-Verlag Berlin Heidelberg.

[89] Song, T., Pan, L., Păun, G. (2013) Asynchronous spiking neural P systems with local synchronization. Information Sciences, vol. 219, pp. 197-207.

[90] Song, T., Pan, L., Jiang, K., Song, B., Chen, W. (2013) Normal Forms for Some Classes of Sequential Spiking Neural P Systems. IEEE Trans. NanoBiosci., vol. 12(3), pp. 1536-1241.

[91] Song, T., Pan, L., Păun, G. (2014) Spiking neural P systems with rules on synapses. Theoretical Computer Science vol. 529(10), pp. 82-95.

[92] Turing, A.M. On Computable Numbers, with an Application to the Entscheidungsproblem. (1936) Proc. London Mathematical Society, Ser. 2, 42, pp. 230-265; a correction (1936), 43, pp. 544-546.

[93] Turing, A.M. (1948) Intelligent Machinery. Report for the National Physical Laboratory, and in (1992) Collected Works of A. M. Turing: Mechanical Intelligence. D.C. Ince (ed.). Elsevier Science Publishers.

[94] von Neumann, J. (1951) The General and Logical Theory of Automata. Cerebral Mechanisms in Behavior: The Hixon Symposium. pp. 1-31, John Wiley & Sons.

[95] Wang, J., Hoogeboom, H.J., Pan, L. (2010) Spiking Neural P Systems with Neuron Division. M. Gheorghe et al. (Eds.): CMC 2010, LNCS 6501, pp. 361-376.

[96] Wang, J., Hoogeboom, H.J., Pan, L., Păun, G., Pérez-Jiménez, M.J. (2010) Spiking Neural P Systems with Weights. Neural Computation, vol. 22(10), pp. 2615-2646.

[97] Wang, J., Zou, L., Peng, H., Zhang, G. (2011) An extended spiking neural P system for fuzzy knowledge representation. J. of Innovative Computing, Information and Control, vol. 7(7A), pp. 3709-3724.

[98] Wang, J., Peng, H., Pérez-Jiménez, M.J., Wang, T. (2013) Weighted Fuzzy Spiking Neural P Systems. IEEE Trans. Fuzzy Sys., vol. 21(2), pp. 209-220.

[99] Zandron, C., Ferretti, C., Mauri, G. (2001) Solving NP-complete problems using P systems with active membranes. In: Antoniou, I. et al (eds.) Unconventional Models of Computation, DMTCS 2001, pp. 289-301.

[100] Zeng, X., Zhang, X., Pan, L. (2009) Homogeneous Spiking Neural P Systems. Fundamenta Informaticae, vol. 97, pp. 1-20.

[101] Zeng, X., Adorna, H.N., Martínez-del-Amor, M.A., Pan, L., Pérez-Jiménez, M.J. . (2011) Matrix Representation of Spiking Neural P Systems. CMC11 LNCS vol. 6501, pp. 377-391.

[102] Zeng, X., Song, T., Pan, L,, Zhang, X. (2012) Performing Four Basic Arithmetic Operations With Spiking Neural P Systems. IEEE Trans. NanoBiosci., vol. 11(4), pp. 366-374.

[103] Zeng, X., Pan, L., Pérez-Jiménez, M.J. (2014) Small universal simple spiking neural P systems with weights. Science China Information Sciences, vol. 57, pp. 1 - 11.

[104] ISI emerging research front, October 2003. In `http://esi-topics.com/erf/october2003.html`

[105] The P systems website. In `http://ppage.psystems.eu/`

Appendix A

Open and online resources for illustrations

- Illustrations of prokaryotic and eukaryotic cells:
  - Cells and DNA, In: Handbook of Genetics, United States National Institutes of Health (2015) accessed: 29 May 2015, http://ghr.nlm.nih.gov/handbook/basics/cell,
  - G. Coté, M. De Tullio. Beyond Prokaryotes and Eukaryotes: Planctomycetes and Cell Organization. Nature vol. 3(9) (2010) accessed: 29 May 2015 http://www.nature.com/scitable/topicpage/beyond-prokaryotes-and-eukaryotes-planctomycetes-and-cell-14158971,
  - Lecture notes and illustrations of Michael Muller, author of "Biology of Cells and Organisms", 5th edition. accessed: 29 May 2015 http://www.uic.edu/classes/bios/bios100/lecturesf04am/lect06.htm
- Illustrations of biological neurons:
  - R.W. Guillery. Observations of synaptic structures: origins of the neuron doctrine and its current status. Philosophical Transactions B of the Royal Society (2005) accessed: 29 May 2015 http://rstb.royalsocietypublishing.org/content/360/1458/1281
  - Scans of the writing and illustrations of neuroscience pioneer and Nobel laureate Santiago Ramón y Cajal, dated 1909 (in French), especially page 152. Histologie du systéme nerveux de l'homme & des vertébrés, por Santiago Ramón y Cajal. In: the Internet Archive. accessed: 29 May 2015 https://archive.org/stream/histologiedusyst01ram#page/152/mode/2up
  - Silvia Helena Cardoso. Neurons: Our Internal Galaxy. accessed: 29 May 2015 http://www.cerebromente.org.br/n07/fundamentos/neuron/rosto_i.htm
  - Silvia Helena Cardoso, Renato M. E. Sabbatini. How Nerve Cells Work. accessed: 29 May 2015 http://www.cerebromente.org.br/n09/fundamentos/transmissao/voo_i.htm

Errata for the dissertation manuscript "Computations in Spiking Neural P Systems: Simulations and Structural Plasticity" by F.G.C. Cabarle

last updated: 28 Feb 2016

1. For Chapter 10, see:

   - F.G.C. Cabarle, N.H.S. Hernandez, M.A. Martínez-del-Amor: Spiking Neural P Systems with Structural Plasticity: Attacking the Subset Sum Problem. Asian Conference on Membrane Computing 2015 (ACMC2015), 12 to 15 November, 2015, Anhui, China, and in LNCS vol 9504, pp. 106-116 (2015) DOI:10.1007978-3-319-28475-0_8