Notes on Spiking Neural P Systems and Finite Automata

Francis George C. Cabarle[1], Henry N. Adorna[1], Mario J. Pérez-Jiménez[2]

[1]Department of Computer Science
University of the Philippines Diliman
Diliman, 1101, Quezon city, Philippines;
[2]Department of Computer Science and AI
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
`fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph, marper@us.es`

Summary. Spiking neural P systems (in short, SNP systems) are membrane computing models inspired by the pulse coding of information in biological neurons. 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. SNP transducers were introduced, where both input and output neurons were used. More recently, SNP modules were introduced 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 work we continue relating SNP modules and finite automata: (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).

Key words: Membrane computing, Spiking neural P systems, Finite automata, Automatic sequences

1 Introduction

Spiking neural P systems (in short, SNP systems) introduced in [7], incorporated into membrane computing the idea of pulse coding of information in computations using spiking neurons (see for example [10][11] and references therein for more information). In pulse coding from neuroscience, pulses known as *spikes* are not distinct, so information is instead encoded in their multiplicity or the time they are emitted.

On the computing side, SNP systems have *neurons* processing only one object (the spike symbol a), and neurons are placed on nodes of a directed graph. Arcs between neurons are called *synapses*. SNP systems are known to be universal in both generative (an output is given, but not an input) and accepting (an input is given, but not an output) modes. SNP systems can also solve hard problems in feasible (polynomial to constant) time. We do not go into such details, and we refer to [7][8][9][16] and references therein.

SNP systems with standard rules (as introduced in their seminal paper) have neurons that can emit at most one pulse (the spike) each step, and either an input or output neuron connected to the environment, but not both. In [15], SNP systems were equipped with both an input and output neuron, and were known as *SNP transducers*. Furthermore, extended rules were introduced in [3] and [14], so that a neuron can produce more than one spike each step. The introduced *SNP modules* in [6] can then be seen as generalizations of SNP transducers: more than one spike can enter or leave the system, and more than one neuron can function as input or output neuron.

In this work we continue investigations on SNP modules. In particular we amend the problem introduced in the construction of [6], 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 [12].

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 generate automatic sequences. Such class of sequences contain, for example, a common and useful automatic sequence known as the Thue-Morse sequence. The Thue-Morse sequence, among others, 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 [1] for further theories and applications related to automatic sequences.

This paper is organized as follows: Section 2 provides our preliminaries. Section 3 provides our results. Finally, section 4 provides our final remarks.

2 Preliminaries

It is assumed that the readers are familiar with the basics of membrane computing (a good introduction is [13] with recent results and information in the P systems webpage[1] and a recent handbook [17]) and formal language theory (available in many monographs). We only briefly mention notions and notations which will be useful throughout the paper.

2.1 Language theory and string notations

We denote the set of natural (counting) numbers as $\mathbb{N} = \{0, 1, 2, \ldots\}$. Let V be an alphabet, V^* is the set of all finite strings over V with respect to concatenation and

[1] `http://ppage.psystems.eu/`

the identity element λ (the empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. We call V a singleton if $V = \{a\}$ and simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. If a is a symbol in V, then $a^0 = \lambda$, A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations union, concatenation, and $+$. Specifically, (i) λ and each $a \in V$ 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. The length of a string $w \in V^*$ is denoted by $|w|$. Unnecessary parentheses are omitted when writing regular expressions, and $E^+ \cup \{\lambda\}$ is written as E^*. We write the language generated by a regular expression E as $L(E)$. If V has k symbols, then $[w]_k = n$ is the base-k representation of $n \in \mathbb{N}$.

2.2 Deterministic finite automata

Definition 1. *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,*
- $F \subseteq Q$ *is a set of final states.*

Definition 2. *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:*

- $Q = \{q_1, \ldots, q_n\}$ *is a finite set of states,*
- $\Sigma = \{b_1, \ldots, b_m\}$ *is the input alphabet,*
- $\Delta = \{c_1, \ldots, c_t\}$ *is the output alphabet,*
- $\delta' : Q \times \Sigma \to Q \times \Delta$ *is the transition function,*
- $q_1 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is a set of final states.*

Definition 3. *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:*

- $Q = \{q_1, \ldots, q_n\}$ *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,*
- $\Delta = \{c_1, \ldots, c_t\}$ *is the output alphabet,*
- $\tau : Q \to \Delta$ *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.*

Definition 4. *A sequence, denoted as* $\mathbf{a} = (a_n)_{n\geq 0}$*, is k-automatic if there exists a k-DFAO, M, such that given* $w \in \Sigma_k^*$*,* $a_n = \tau(\delta''(q_1, w))$*, where* $[w]_k = n$*.*

Example 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. 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 $t_3 = 0$.

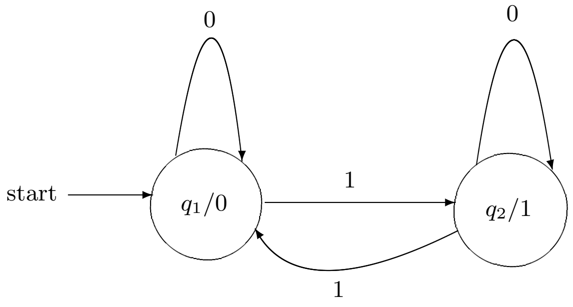

Fig. 1. 2-DFAO generating the Thue-Morse sequence.

2.3 Spiking neural P systems

Definition 5. *A spiking neural P system (in short, an SNP 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: $E/a^c \to a^p; d$, 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 $E/a^c \to a^p; d$ 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 $a^c \to \lambda$, known as a *forgetting rule.* If a spiking rule has $L(E) = \{a^c\}$, we simply write it as $a^c \to a^p; d$. The systems from the original paper [7], with rules of the form $E/a^c \to a; d$ and $a^c \to \lambda$, are referred to as *standard* systems with *standard rules.* The extended rules (i.e. $p \geq 1$) used in this work are referred to as SNP systems with extended rules in other literature, e.g. [6], [14], [16].

The rules are applied as follows: 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 (also referred to as *spiking*) after d time units, to every σ_j where $(i, j) \in syn$. 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 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 $E_1/a^{c_1} \rightarrow a^{p_1}; d_1$ and $E_2/a^{c_2} \rightarrow a^{p_2}; d_2$, 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.

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 $r_\ell = E_\ell/a^{j_\ell} \rightarrow a^{p_\ell}; d_\ell, 1 \leq \ell \leq m$, in σ_{out}. From [3] 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 [6], we only consider the latter.

Definition 6. *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: $E/a^c \rightarrow a^p$, 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*);

- $N_{in}, N_{out} (\subseteq \{1, 2, \dots, m\})$ indicate the *sets of input* and *output* neurons, respectively.

In [15], SNP transducers operated on strings over a binary alphabet as well considering b_0 as a symbol. SNP modules, first introduced in [6], are a special type of SNP systems with extended rules, and 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 (in short, Env); Each output neuron $\sigma_o, o \in N_{out}$, produces p spikes to Env, 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.

3 Main results

In this section we amend and improve constructions given in [6] to simulate DFA and DFST using SNP modules. Then, k-DFAO are also simulated with SNP modules. Lastly, SNP modules are related to k-automatic sequences.

3.1 DFA and DFST simulations

We briefly recall the constructions from theorem 8 and 9 of [6] for SNP modules simulating DFAs and DFSTs. The constructions for both DFAs and DFSTs have a similar structure, which is shown in Fig. 2. For neurons 1 and 2 in Fig. 2, 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 $\Sigma = \{b_1, \dots, b_m\}$, $Q = \{q_1, \dots, q_n\}$. The construction for theorem 8 of [6] for an SNP Module Π_D simulating D is as follows:

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

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 Fig. 2. 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 Env, 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 $w = b_{i_1}, \dots, b_{i_r}$ in Σ^* and produces a sequence of states $z = q_{i_1}, \dots, q_{i_r}$ (represented by $a^{i_1}, \dots, a^{i_r}$) such that $\delta(q_{i_\ell}, b_{i_\ell}) = q_{i_{\ell+1}}$, for each $\ell = 1, \dots, r$ 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 [6]

Theorem 1. *(Ibarra et al [6]) Any regular language L can be expressed as $L = L(\Pi_D)$ for some SNP module Π_D.*

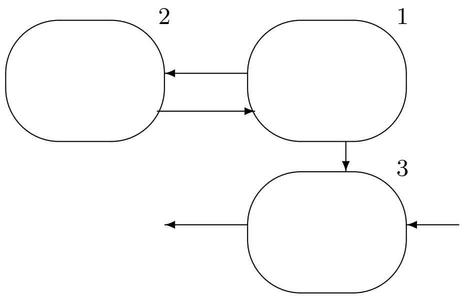

Fig. 2. Structure of SNP modules from [6] simulating DFAs and DFSTs.

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 $\Sigma = \{b_1, \dots, b_k\}$, $\Delta = \{c_1, \dots, c_t\}$, $Q = \{q_1, \dots, q_n\}$. 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 Fig. 2. 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}, 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 1, the following is Theorem 9 from [6]:

Theorem 2. *(Ibarra et al[6]) Any finite transducer T can be simulated by some SNP module Π_T.*

The previous constructions from [6] 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 2.

Example 2. We modify the 2-DFAO in Fig. 1 into a DFA in Fig. 3 as follows: Instead of $\Sigma = \{0, 1\}$, we have $\Sigma = \{1, 2\}$; We maintain $n = m = 2$, however, the transitions are swapped, so in Fig. 3 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 Fig. 4. 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 a^3 from Env (representing input 1 for the DFA), at the next step σ_3 will have a^7 spikes, allowing the possibility of an incorrect simulation.

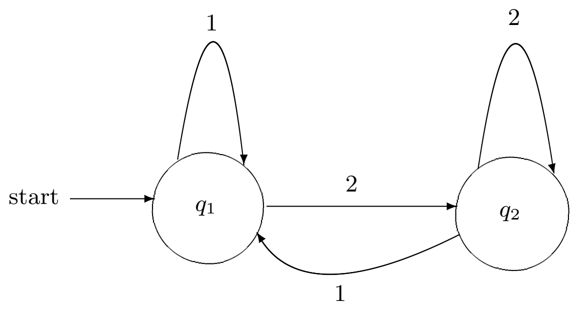

Fig. 3. DFA with incorrect simulation by the SNP module in Fig. 4.

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:

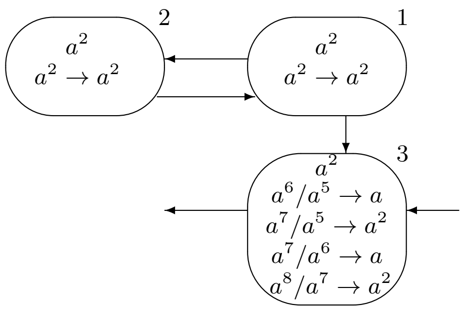

Fig. 4. SNP module with incorrect simulation of the DFA in Fig. 3.

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

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 1 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 Env 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 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

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

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

3.2 *k*-DFAO simulation and generating automatic sequences

Next, we modify the construction from Theorem 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-DFAOs from Definition 3 and DFSTs from Definition 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, 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 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:

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

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 Env. 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 5. *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. *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. [1]) 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 5 simulates both δ'' and τ, we can include robustness properties as the following result shows.

Theorem 7. *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 5 is given in example 3.

Example 3. (SNP module simulating the 2-DFAO generating the Thue-Morse sequence) The SNP module is given in Fig. 5, and we have $n = m = t = 2$. Based on the construction for Theorem 5, we associate 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 Fig. 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.

4 Final Remarks

In [3], strict inclusions for the types of languages characterized by SNP systems with extended rules having one, two, and three neurons were given. Then in [15], it was shown that there is no SNP transducer that can compute nonerasing and

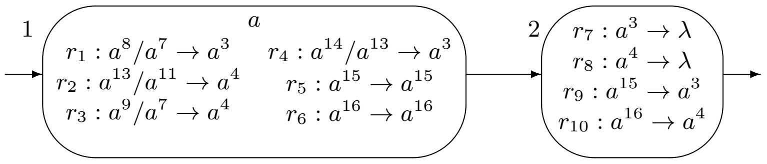
1
a
$r_1 : a^8/a^7 \to a^3$
$r_2 : a^{13}/a^{11} \to a^4$
$r_3 : a^9/a^7 \to a^4$
$r_4 : a^{14}/a^{13} \to a^3$
$r_5 : a^{15} \to a^{15}$
$r_6 : a^{16} \to a^{16}$
2
$r_7 : a^3 \to \lambda$
$r_8 : a^4 \to \lambda$
$r_9 : a^{15} \to a^3$
$r_{10} : a^{16} \to a^4$

Fig. 5. SNP module simulating the 2-DFAO in Fig. 1.

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 [1]) 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 [15] a time step without a spike entering or leaving the system was considered as a symbol of the alphabet, while in [6] (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 3 to 4, SNP modules consist of only one neuron for each module, compared to three neurons in the constructions of [6]. 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 [6]). On the practical side, SNP modules might also be used for computing functions, as well as other tasks involving (streams of) input-output transformations. Practical applications might include image modification or recognition, sequence analyses, online algorithms, et al.

Some preliminary work on SNP modules and morphisms was given in [2]. From finite sequences, it is interesting to extend SNP modules to infinite sequences. In [4], extended SNP systems[2] 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 [5]. While the syntax of SNP modules may prove sufficient for these "interactive tasks", or at least only minor modifications, a (major) change in the semantics is probably necessary.

Acknowledgements

Cabarle is supported by a scholarship from the DOST-ERDT of the Philippines. Adorna is funded by a DOST-ERDT grant and the Semirara Mining Corp. professorial chair of the College of Engineering, UP Diliman. M.J. Pérez-Jiménez acknowledges the support of the Project TIN2012-37434 of the "Ministerio de

[2] or ESNP systems, in short, are generalizations of SNP systems almost to the point of becoming tissue P systems. ESNP systems are thus generalizations also of (and not to be confused with) SNP systems with extended rules.

Economía y Competitividad" of Spain, co-financed by FEDER funds. Fruitful discussions with Miguel Ángel Martínez-del Amor are also acknowledged.

References

1. Allouche, J-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)
2. Cabarle, F.G.C, Buño, K.C., Adorna, H.N.: Spiking Neural P Systems Generating the Thue-Morse Sequence. Asian Conference on Membrane Computing 2012 pre-proceedings, 15-18 Oct, Wuhan, China (2012)
3. Chen, H., Ionescu, M., Ishdorj, T-O., Păun, A., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with extended rules: universality and languages. Natural Computing, vol. 7, pp. 147-166 (2008)
4. Freund, R., Oswald, M.: Regular ω-languages defined by finite extended spiking neural P systems. Fundamenta Informaticae, vol. 81(1-2), pp. 65-73 (2008)
5. Goldin, D., Smolka, S., Wegner, P. (Eds.): Interactive Computation: The New Paradigm. Springer-Verlag (2006)
6. Ibarra, O., Peréz-Jiménez, M.J., Yokomori, T.: On spiking neural P systems. Natural Computing, vol. 9, pp. 475-491 (2010)
7. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae, vol. 71(2,3), pp. 279-308 (2006)
8. Leporati, A., Zandron, C., Ferretti, C., Mauri, G.: Solving Numerical NP-Complete Problems with Spiking Neural P Systems. Eleftherakis et al. (Eds.): WMC8 2007, LNCS 4860, pp. 336-352 (2007)
9. Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China Information Sciences. vol. 54(8) pp. 1596-1607 (2011)
10. Maass, W.: Computing with spikes, in: Special Issue on Foundations of Information Processing of TELEMATIK, vol. 8(1) pp. 32-36, (2002)
11. Maass, W., Bishop, C. (Eds.): Pulsed Neural Networks, MIT Press, Cambridge (1999)
12. Neary, T.: A Boundary between Universality and Non-universality in Extended Spiking Neural P Systems. LATA 2010, LNCS 6031, pp. 475-487 (2010)
13. Păun, Gh.: Membrane Computing: An Introduction. Springer-Verlag (2002)
14. Păun, A., Păun, Gh.: Small universal spiking neural P systems. BioSystems, vol. 90(1), pp. 48-60 (2007)
15. Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G.: Computing Morphisms by Spiking Neural P Systems. Int'l J. of Foundations of Computer Science. vol. 8(6) pp. 1371-1382 (2007)
16. Păun, Gh., Pérez-Jiménez, M.J.: Spiking Neural P Systems. Recent Results, Research Topics. A. Condon et al. (eds.), Algorithmic Bioprocesses, Springer-Verlag (2009)
17. Păun, Gh., Rozenberg, G., Salomaa, A. (Eds) The Oxford Handbook of Membrane Computing, OUP (2010)