A Model Learning based Testing Approach for Spiking Neural P Systems

Florentin Ipate[1] and Marian Gheorghe[2]

[1] *Department of Computer Science*
Faculty of Mathematics and Computer Science
University of Bucharest
Str. Academiei 14, Bucharest 010014, Romania
florentin.ipate@unibuc.ro

[2] *Department of Computer Science*
University of Bradford
Bradford BD7 1DP, UK
m.gheorghe@bradford.ac.uk

Abstract

This paper proposes a testing approach for spiking neural P systems, significantly different from the past testing research for cell-like P systems. The proposed method provides a solution to the state explosion problem by constructing a series of approximations, using the concept of cover automaton and Angluin-style model learning from queries, more precisely the L^l algorithm for learning a finite cover automaton, adapted to the more general X-machine model. Furthermore, the concept of idenfiability, which is an essential prerequisite for the successful application of our method, but also a more general design characteristic inspired from the testing practice, is introduced and investigated in the context of spiking neural P systems. Identifiability of system's components (modules, methods, etc.) is a fundamental criterion used for assessing a system's testability since it allows the components of a system to be identified from the behaviour produced in response to the inputs received and, consequently, maximizes the effectiveness of the testing process.

Keywords: spiking neural P systems, testing, finite automata, model learning

1. Introduction

Membrane computing is a branch of natural computing inspired by the structure and functioning of the living cells. This computation paradigm was introduced in [34] and the models have been called membrane systems or P systems. Due to a rapid and sustained development, an initial research monograph [35] and, later on, a comprehensive handbook, covering both theoretical results and applications [36], were published. A specific model of membrane computing, called spiking neural P system, is inspired by the neuron cells. The model was introduced in [21] and has been intensively investigated. A survey paper [38] presents the key theoretical developments in this area and the main applications of the model. Some of the most relevant applications of spiking neural P systems are in modelling arithmetic operations [30, 43, 29, 41, 45], error-tolerant serial binary full-adder [31], skeletonizing images [14], solving combinatorial optimizations [44]. Fuzzy neural P systems are applied in solving fault diagnosis problems [39, 42]. A special line of research in membrane systems (many of them with respect to neural P systems) is dedicated to the formal analysis of these systems. This includes formal semantics [5, 7, 12, 13], reversible computation [1, 37, 3], causality [2, 32] and memory associated with these systems [11]. Testing, especially model-based approach using membrane systems, is another type of formal analysis of these systems looking at traces of execution (computation pathways) defined with respect to certain formal principles. This approach is also significant for validating applications of membrane systems. Testing is the process of running a software system or program with the purpose of discovering bugs. Testing is the main validation technique used in the industry. Even when the model of the system has been formally verified, as it is the case with safety critical applications, the system is still tested; this is because the behaviour of the actual system *implementation* may differ from the verified model. Since even for trivial programs exhaustive testing (i.e., running the program on all its possible inputs) is impossible or at least impractical, test selection (or test generation) is a major part of testing and various techniques exist. Essentially,

these seek to "cover" as much as possible of the program specification and/or implementation so as to maximise the likelihood of fault detection when these tests are run.

A major class of test generation techniques are grouped under the name of black-box testing. In black-box testing (also called functional testing), tests are derived from the requirements (or specification) and the *implementation (or system) under test* is regarded as a black-box: tests are applied, and results can be observed but nothing is known about the structure of implementation (which is a black-box). In many cases in the software industry requirements are expressed informally and so the test generation techniques are also informal. On the other hand, when a formal model exists, test data may be selected in a more rigorous way so that a certain level of coverage or fault detection is achieved. Also, the existence of a model allows test generation to be automated, which is a big plus when it comes to the practical applicability of such techniques. Black-box testing in the presence of a formal model is called *model-based testing*. Usually, state-based models, composed of states and transitions between states, are used. The most important limitation of model-based testing is related to the size of the model produced: as the number of state variables increases, the number of states grows exponentially. This is called the state explosion problem. Hence, a major challenge of model-based testing is devising techniques to reduce or alleviate this problem.

The major limitation of (black box) testing, seen from the perspective of the formal methods community, is that "program testing can be used to show the presence of bugs, but never to show their absence!" (Edsger W. Dijkstra). This is obviously true when tests are derived from an informal specification, but also, to some extent in model-based testing since most test generation techniques are aimed at achieving a certain coverage level of the model, which is not directly linked with the absence of the errors. On the other hand, there are model-based testing techniques that guarantee complete fault-detection, albeit under certain (more or less restrictive) assumptions. Suppose M is the model of the system and I the implementation under test. Naturally, in black-box testing I is not

known, but one can define a *fault model*, a set C of models such that, if the model of the implementation under test I belongs to C and I passes all tests, I is guaranteed to be fault-free (its model is functionally equivalent to M). One well-known example of such techniques is the W-method [10]. Here, the model is a finite state machine (more precisely, the Mealy machine variant) and the fault model is, for a given $k \geq 0$, the set of all Mealy machines whose number of states does not exceed the number of states n of M by more than k. The downside is that the size of the test suite generated by the method is exponential in k. However, when the implementation is relatively close to the model, k is normally relatively low, and so the total size of the test suite is a polynomial of low degree (in the size of the input alphabet and the size of the state set).

Model based testing approaches have been introduced and studied for cell-like P systems [16, 24, 17]. Also, mutation testing for the same class of P systems has been considered [25]. However, although important applications of spiking neural P systems exist, to the best of our knowledge, testing of these models has not been approached yet.

This paper proposes a testing method for spiking neural P systems. The underlying test generation strategy is based on a generalization of the W-method (for this, the spiking neural P system is transformed into a type of extended finite state machine called X-machine) and so the method guarantees total fault detection under the assumptions of the W-method. Testing of a P system model has been discussed in the past, but this paper makes the following significant advances.

- It addresses the testing of spiking neural systems in a context different from that utilised for cell-like P systems.
- It offers a solution to the state explosion problem by constructing a series of approximations, using the concept of *cover automaton*. In order to construct these approximations, Angluin's learning from queries [6] and the L^l algorithm for learning a finite cover automaton [23] are used. The algorithm is adapted to the more general X-machine model.

– It investigates the concept of identifiability in the context of spiking neural P systems, which is an essential prerequisite for the successful application of our method. In general, identifiability of system's components (modules, methods, etc.) is a fundamental criterion used for assessing a system's testability since it allows the components of a system to be identified from the behaviour produced in response to the inputs received; identifiability is also an essential ingredient in a "design for test" strategy, in which a system is designed so as to maximize the effectiveness of the testing process [19].

For simplicity, in this paper the proposed method is presented for spiking neural P systems without delays but the approach can be naturally extended to spiking neural P systems with delay rules.

The paper is structured as follows. The main concepts and results to be used in this paper are presented in sections 2 (spiking neural P systems, finite automata, finite cover automata Mealy machines), 3 (the W-method for cover automata) and 4 (the L^l algorithm for learning cover automata). The remainder of the paper presents the proposed method and its technical underpinnings: X-machines and X-machine based testing are introduced in section 5, the application of the forementioned testing and learning results to spiking neural P systems is discussed in section 6, while section 7 investigates identifiability in spiking neural P systems. The next section discusses practical details for the application of our method. Finally, conclusions are drawn and future work is outlined in section 9.

2. Preliminaries

In this section, we introduce the main theoretical concepts and models to be used in this paper: spiking neural systems and finite state machines models, namely finite automata, Mealy machines and cover automata.

In what follows, for a finite alphabet $V = \{a_1, ..., a_p\}$, V^* denotes the set of all strings (sequences) over V; the empty string is denoted by ϵ. The length

(number of characters) of a string u is denoted by $length(u)$; $length(\epsilon) = 0$. V^n denotes the set of all strings of length n, $n \geq 0$, with members in the alphabet V and $V[n] = \bigcup_{0 \leq i \leq n} V^i$. The language described by a regular expression E is denoted by $L(E)$. For a finite set A, $card(A)$ denotes the number of elements of A.

2.1. Spiking neural P systems

We now introduce the spiking neural P system model. The definition related concepts presented here are largely from [21] and [33].

Definition 1. A *spiking neural P system* (abbreviated *SN P system*) of degree m, $m \geq 1$, is a tuple $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$, where

- $O = \{a\}$ is a singleton alphabet (a is called spike);
- $\sigma_i, 1 \leq i \leq m$, are neurons, $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where
  - $n_i \geq 0$ is the number of spikes in σ_i;
  - R_i is a finite set of rules of the following forms:
    - (*Type (1); spiking rules*)
      $E/a^c \rightarrow a^p$; where E is a regular expression over $\{a\}$, and $c \geq 1$, $p \geq 1$, such that $c \geq p$;
    - (*Type (2); forgetting rules*)
      $a^s \rightarrow \epsilon$, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a^p$ of type (1) from R_i, $a^s \notin L(E)$;
- $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn$, $1 \leq i, j \leq m$ (synapses between distinct neurons);
- $in, out \in \{1, \ldots .m\}$ indicate the input and output neurons respectively.

An SN P system computes by applying one rule from each neuron at a time. A rule of type (1) $E/a^c \rightarrow a^p$ can be applied if there are n spikes in neuron σ_j, $n \geq c$ and $a^n \in L(E)$. In defining type (1) rules in SN P systems, we follow

the standard convention of not specifying E whenever the left-hand side of the rule is equal to E. Note that, in general, there may be a delay between the time when a type (1) rule fires and the time the spike is emitted. However, in this paper we consider only SN P systems without delay rules.

A rule of type (2) $a^s \to \epsilon$ removes spike(s) from the neuron; this can only be applied if the number of spikes in σ_j is exactly s, the number of spikes it needs to be applied.

In general, it is possible to have $a^n \in L(E_x) \cap L(E_y)$, for some rules of type 1, r_x and r_y, $x \neq y$, in neuron σ_j. In this case one of the two rules that can be applied will be chosen. This is how the non-determinism of the SN P system is realised. On the other hand, when a rule of type (2) is applicable, no other rule is applicable in the same neuron. Since rules from several neurons can fire (or spike) simultaneously, the system also exhibits parallelism.

A *configuration* of an SN P system Π is an m-tuple of integers $C = (a_1, a_2, \ldots, a_m)$, where a_j, $1 \leq j \leq m$, represents the number of spikes in neuron σ_j. The vector $C^{(0)} = (a_1^{(0)}, a_2^{(0)}, \ldots, a_m^{(0)})$, where $a_j^{(0)}$, $1 \leq j \leq m$, defines the initial number of spikes in neuron σ_j, represents the *initial configuration* of Π. A configuration for which no rule can be applied in any of the system's compartments is called a *halting configuration*. The sequence of configurations $C^{(0)}C^{(1)} \ldots C^{(n)}$ starting with the initial configuration is called a *computation* of the system.

An SN P system obtains inputs from the environment through the designated input neuron, while the result produced can be observed through the designated output neuron. One way in which the input is supplied to the system is in the form of k natural numbers. More precisely, the SN P system reads from the environment a binary sequence $z = 10^{n_1-1}0^{n_2-1}1 \ldots 10^{n_k-1}$; the input neuron receives a spike in each step corresponding to a 1 and no spike otherwise, until all digits in the input sequence have been consumed [33]. Depending on the purpose of the system, the output of the system can be represented in various ways (see [21] and [33]). In this paper, however, we will not distinguish the output neuron and, furthermore, we will consider that all system's neurons can be observed; this is consistent with the testing practice of observing the values

all system's variables in any of its state.

2.2. Finite automata

A finite state machine is a computing device composed of a finite number of states and transitions between states labelled by symbols. In this paper, two finite state machine variants will be used: finite automata and Mealy machines. In a finite automaton (FA) the transitions are labelled by mere inputs, whereas the transitions of a Mealy machine are labelled by input-output pairs. Essentially, finite automata are finite state acceptors while Mealy machines are transducers.

Definition 2. A *deterministic finite automaton* (abbreviated *DFA*) is a tuple $A = (\Sigma, Q, q_0, F, \delta)$, where:

- Σ is the finite *input alphabet*;
- Q is the finite *set of states*;
- $q_0 \in Q$ is the *initial state*;
- $F \subseteq Q$ is the *set of final states*;
- $\delta : Q \times \Sigma \longrightarrow Q$ is the *next-state function*.

The next-state function δ can be extended to take sequences of inputs, i.e., $\delta : Q \times \Sigma^* \longrightarrow Q$ [15]. Given $q \in Q$, the set $L^q(A)$ is defined by $L^q(A) = \{s \in \Sigma^* \mid \delta(q, s) \in F\}$. When q is the initial state of A, the set is called the *language accepted by* A and the simpler notation $L(A)$ is used.

A language that is accepted by some deterministic finite automaton is called a *regular language*. Given a regular language L, a DFA that accepts L is called *minimal* if any other DFA that accepts L has more states than A. A minimal DFA that accepts a given regular language L is unique (up to a renaming of the state set) [20]. Given a DFA A, a minimal DFA that accepts $L(A)$ can be constructed by removing the states that cannot be reached with appropriate input sequences from the initial state and by "merging" all states that accept

identical sets of inputs. More formally, a state $q \in Q$ is called *reachable* if there exists $\sigma \in \Sigma^*$ such that $\delta(q_0, \sigma) = q$. Given $Y \subseteq \Sigma^*$, two states $q_1, q_2 \in Q$ are called Y-*equivalent* if $L^{q_1}(A) \cap Y = L^{q_2}(A) \cap Y$. Otherwise q_1 and q_2 are called Y-*distinguishable*. States q_1 and q_2 are called *distinguishable* if they are Y-distinguishable for some $Y \subseteq \Sigma^*$. Then, a DFA A is minimal if and only if (1) every state is reachable and (2) any two distinct states are distinguishable [20].

2.3. Finite cover automata

In some practical applications, input sequences that exceed a certain length are never used. In this case, the problem of constructing a (minimal) automaton can be reformulated as follows: given a finite language U and l the length of the longest sequence(s) in U, construct a (minimal) FA that accepts all sequences in U and rejects all sequences in $\Sigma[l] \setminus U$, but may accept or reject all remaining sequences (the behaviour of the device for longer sequences is not relevant). Such a device is called a finite cover automaton [8, 9].

Definition 3. Let $A = (\Sigma, Q, q_0, F, \delta)$ be a FA, $U \subseteq \Sigma^*$ a finite language and l the length of the longest sequence(s) in U. Then A is called a *deterministic finite cover automaton* (*DFCA*) of U if $L(A) \cap \Sigma[l] = U$.

A *minimal* DFCA for U is a DFCA for U having the least number of states. A minimal DFCA for U may have much less states than the minimal DFA that accepts U [22] and so, in applications in which the longer sequences are not used, it is preferable to construct a minimal DFCA instead of the minimal DFA that accepts U.

Similarly to the minimal DFA, a minimal DFCA is constructed by removing the unreachable states and merging the states that exhibit "similar" behaviour. Unlike in DFA minimisation though, the similarity relation on the state set used in this case is not necessarily an equivalence relation and so the induced decomposition on the state set is not necessarily an equivalence relation. Consequently, there may exist more than one minimal deterministic finite cover automaton of

the same finite language U. For further details, the reader is referred to [8] and [28].

2.4. Mealy machines

Unlike finite automata, the transitions of Mealy machines are labelled by input-output pairs.

Definition 4. A *Mealy machine* is a tuple
$M = (\Sigma, \Gamma, Q, q_0, \delta, \lambda)$, where:

- Σ is the finite *input alphabet*;
- Γ is the finite *output alphabet*;
- Q is the finite *set of states*;
- $q_0 \in Q$ is the *initial state*;
- $\delta : Q \times \Sigma \longrightarrow Q$ is the *next-state function.*
- $\lambda : Q \times \Sigma \longrightarrow \Gamma$ is the *output function.*

Naturally, the next-state and output functions can be extended to take sequences and produce the destination state and corresponding output function, respectively, i.e., $\delta : Q \times \Sigma^* \longrightarrow Q$ and $\lambda : Q \times \Sigma^* \longrightarrow \Gamma^*$ [15]. The input-output mapping from state $q \in Q$ is denoted by λ_q, i.e., $\lambda_q(s) = \lambda(q, s), \forall s \in \Sigma^*$.

The two finite state machine formalisms share many similarities. A deterministic FA A can be naturally rewritten as a Mealy machine M. For instance, two output symbols 0 and 1 can be introduced ($\Gamma = \{0, 1\}$); 0 will be associated with transitions in A leading to a final state and 1 with transitions leading to a non-final state. Then, for every input sequence $s \in \Sigma^* \setminus \{\epsilon\}$, $s \in L$ if and only if the last symbol of the corresponding output sequence $\lambda_{q_0}(s)$ is 0. On the other hand, state distinguishability can also be defined in the context of Mealy machines by examining the outputs produced, i.e., given $Y \subseteq \Sigma^*$, two states $q_1, q_2 \in Q$ are said to be *Y-distinguishable* if $\lambda_{q_1}(s) \neq \lambda_{q_2}(s)$ for some

$s \in Y$. Similarly to finite automata, the minimality problem for Mealy machines is defined and addressed using state reachability and distinguishability [15]. Furthermore, analogously to cover automata, an l-minimal Mealy machine is defined for any $l \geq 1$; this is a Mealy machine M having the least number of states among all Mealy machines M' that produce identical output sequences in response to any input sequence no longer than the upper bound l. Like in the case of finite automata, the minimal Mealy machine (for a given machine) is always unique [15], while many l-minimal Mealy machines may exist [22].

3. Bounded sequence testing from Mealy machines

Suppose our system is modelled by a Mealy machine and we would like to test its unknown implementation. The W-method [10] assumes that it is possible to estimate the maximum number of states of the implementation and generates a test suite that, if applied to the implementation and all tests pass, will guarantee that the implementation is functionally equivalent to the model. More precisely, if C_k is the set of all Mealy machines with the same input and output alphabets as M having no more than k more states than M, $k \geq 0$, the W-method will produce a finite set X_k of input sequences such that, for every $M' \in C_k$, M and M' will produce identical output sequences on every sequence in X_k if and only if M and M' are functionally equivalent (they produce identical output sequences in response to any input sequence).

More recently, W-method has been adapted to generate test suites from finite cover automata [22]. Here, given an upper bound l, the generated test suite Y_k will have the property that, for every $M' \in C_k$, M and M' will produce identical output sequences on every sequence in Y_k if and only if M and M' produce identical output sequences in response to any sequence of up to l elements. Naturally, Y_k may be the set of all sequences in $\Sigma[l]$, but usually, since l is fairly large, this is impractical. The W-method, on the other hand, will produce test suites of a much lower size. The technical details of the method, largely from [22], are given below.

Let $M = (\Sigma, \Gamma, Q, q_0, \delta, \lambda)$ be a Mealy machine. Given a state q of M, $level_M(q)$ is defined to be the length of the shortest input sequence(s) that reach q, i.e., if $\delta(q_0, s) = q$ then $length(s) \geq level_M(q), \forall s \in \Sigma^*$. Let $l \geq 1$ and suppose M is l-minimal (this does not restrict the applicability of the method since minimization algorithms, such as the algorithm provided in [27], can be applied beforehand). Then a proper state cover and a characterisation set, as defined next, exist.

Definition 5. $S \subseteq \Sigma^*$ is called a *proper state cover* of M if, for every state $q \in Q$, there exists $s \in S$ such that $\delta(q_0, s) = q$ and $length(s) = level(q)$.

Definition 6. $W \subseteq \Sigma^*$ is called a *strong characterisation set* of M, if for every two states $q_1, q_2 \in Q$ and every $j \geq 0$, if q_1 and q_2 are $\Sigma[j]$-distinguishable then q_1 and q_2 are $(W \cap \Sigma[j])$-distinguishable.

Note that, in the above definition, it is sufficient for the implication to hold when j is the length of the shortest sequence that distinguishes between q_1 and q_2.

Then, as proved in [22], a test suite for M with respect to the fault model consisting of all Mealy machines having no more than k extra states than M, $k \geq 0$, can be constructed using the formula:

$$Y_k = S\Sigma[k+1](W \cup \{\epsilon\}) \cap \Sigma[l] \setminus \{\epsilon\}.$$

Therefore a test suite can be obtained from the concatenation of three sets: a proper state cover S, the set $\Sigma[k+1]$ of all input sequences of length up to $k+1$ and a a strong characterisation set W to which the empty sequence ϵ is added. Naturally, only sequences of length less than or equal to l are extracted from the resulting set and the empty sequences is also removed.

Theorem 1. *Let $l \geq 1$ and $M = (\Sigma, \Gamma, Q, q_0, \delta, \lambda)$ be an l-minimal Mealy machine. Let S and W be a proper state cover and a strong characterisation set of M, respectively. Then, for any $k \geq 0$ and any Mealy machine $M' = (\Sigma, \Gamma, Q', q_0', \delta', \lambda')$ such that $card(Q') - card(Q) \leq k$, the following holds: $\lambda_{q_0}(s) = \lambda'_{q_0'}(s), \forall s \in \Sigma[l]$ if and only if $\lambda_{q_0}(t) = \lambda'_{q_0'}(t), \forall t \in Y_k$.*

As a finite automaton can be rewritten as a Mealy machine, the above result will be used in the remainder of the paper to test deterministic (cover) automata and their generalizations, X-machines.

4. Learning cover automata from queries

Learning finite automata from queries was introduced by Angluin in 1987 [6]; the paper also provides an algorithm, called L^* for addressing this issue. The setup is as follows: a *learner* seeks to construct a (minimal) deterministic finite automaton for an unknown regular language L by repeatedly asking queries to a *teacher* and an *oracle*. Two types of queries can be addressed. First, given an input sequence s, the learner may ask if $s \in L$; this is called a *membership query*. On the basis of the responses received, the L^* produces a candidate automaton. At this point, the learner may ask the oracle if this is the correct automaton, i.e., if it accepts the regular language L; this is called an *equivalence query*. If the response is negative, the oracle will also provide a counterexample, that will be used by the algorithm to start the construction of a new candidate automaton. Using both membership and equivalence queries, the L^* will construct a minimal deterministic finite automaton A for the language L in polynomial number of steps in the number of states of A. The algorithm has been adapted to construct a minimal deterministic finite cover automaton of a finite language U [23]. The resulting algorithm, called L^l is described next.

Similarly to L^*, the L^l algorithm uses two types of queries: *membership queries* and *language queries*. Let $l > 0$ be length of the longest sequence in U. The L^l algorithm constructs two sets: a non-empty, prefix-closed, set of input sequences $S \in \Sigma^*$ and a non-empty, suffix-closed, set of input sequences $W \in \Sigma^*$. S is used to reach the states and W to distinguish between the states of the candidate automaton. S and W will be constructed such that S will only contain sequences of up to l inputs, while W will only contain sequences of up to $l - 1$ inputs.

The algorithm also keeps an *observation table*. The rows in the table are

labelled by the elements of $(S \cup S\Sigma) \cap \Sigma[l]$, while the columns are labelled by the elements of W and each element in the table is formed from the concatenation of the row label $s \in (S \cup S\Sigma) \cap \Sigma[l]$ and column label $w \in W$ and is assigned one of the values 0, 1 or -1. Thus, the observation table can be formally described by a mapping $T : ((S \cup S\Sigma) \cap \Sigma[l])W \longrightarrow \{0, 1, -1\}$. For $s \in (S \cup S\Sigma) \cap \Sigma[l]$ and $w \in W$, the value of $T(sw)$ is established through the membership query: if $sw \in U$ then $T(sw) = 1$; if $u \in \Sigma[l] \setminus U$ then $T(sw) = 0$; if sw is longer than l ($sw \notin \Sigma[l]$), the response produced by the candidate automaton to be constructed is not relevant, so $T(sw)$ is given a third value, -1, i.e., $T(sw) = -1$.

In order to compare the rows in the observation table, a relation on these rows, called *similarity*, is defined. Given, k, $1 \leq k \leq l$, rows s and s' are said to be *k-similar*, written $s \sim_k s'$ if, for every $w \in W$ with $length(w) \leq k - max\{length(s), length(s')\}$, $T(sw) = T(s'w)$. That is, all corresponding values sw and sw' in rows s and s' coincide for every column w for which sw and $s'w$ have at most k elements. Otherwise, s and s' are said to be *k-dissimilar*, written $s \nsim_k s'$. Using this similarity relation, two properties of an observation table are defined: consistency and closedness. The observation table is said to be *consistent* if, for every k, $1 \leq k \leq l$, whenever rows $s \in S$ and $s' \in S$ are k-similar, rows $s\sigma$ and $s'\sigma$ are also k-similar for all $\sigma \in \Sigma$. The observation table is said to be *closed* if, for any row $s \in S\Sigma$, there exists row $s' \in S$ with $length(s') \leq length(s)$, such that $s \sim s'$.

Initially, $S = W = \{\epsilon\}$. The L^l algorithm periodically checks the consistency and closedness properties of the observation table. If the observation table is not consistent then a suitable new column is added; similarly, a new row is added when the table is found not to be closed. When both conditions are met, a *candidate* DFA, denoted $A(S, W, T)$ is constructed from the consistent and closed observation table. A language query is then addressed to the oracle; if the candidate DFA is a cover automaton of $A(S, W, T)$ (i.e., the language L accepted by $A(S, W, T)$ satisfies $L \cap \Sigma[l] = U$) then the algorithm ends; otherwise, the counterexample produced by the oracle is used by L^l to start a new iteration and produce a new candidate DFA. The pseudocode description

of the L^l algorithm between two language queries is given in Figure 1.

Given a consistent and closed observation table, the candidate DFA $A(S, W, T)$ is defined as follows. Given $s \in S \cup S\Sigma$, the minimum sequence $t \in S$ according to the quasi-lexicographical order on Σ^* for which $s \sim_l t$ is denoted by $r(s)$ (in particular $r(\epsilon) = \epsilon$). Then $A(S, W, T) = (\Sigma, Q, q_0, F, \delta)$, where

- $Q = \{r(s) \mid s \in S\}$;
- $q_0 = \epsilon$;
- $F = \{t \mid t \in Q, T(t) = 1\}$;
- $\delta(t, \sigma) = r(t\sigma)$ (the consistency and closedness properties ensure that δ is well-defined [23]).

Using both membership and language queries, the L^l algorithm will find a minimal DFCA of U in polynomial time. For more details, we refer the reader to [23].

5. Modelling and testing using X-machines

In this paper, the X-machine will be used as a vehicle for applying the previously presented modelling and testing techniques to spiking neural P systems: an SN P system will be transformed into an X-machine using the learning algorithm presented in section 4; then the testing technique described in section 3 will be used to generate tests from the X-machine model.

An X-machine is a type of extended finite state machine (extended automaton) in which transitions between states are labelled by partial functions operating on a data set X. The concept was originally introduced by Eilenberg [15]. Subsequently, a variant, called *stream X-machine*, in which the data set is composed of an input and an output stream of symbols and an internal memory storage, was mostly used [19, 18]. In here, however, we use the general form of the data set X. In order to use the X-machine as a modelling tool for spiking neural P systems, we slightly modify its original definition in that the machine

may use a set of initial data values instead of only one. Such a construct was referred by Eilenberg as an X-module [15]; however, in this paper we keep the more well-known name of X-machine.

We now introduce the X-machine model and preliminary results to be used for the testing approach developed in the next section; essentially, we revisit the results given in [17] to allow for the more general definition of an X-machine used in this paper.

Definition 7. An *X-Machine* (abbreviated *XM*) is a tuple $Z = (X, \Phi, X_0, Q, q_0, \delta)$, where:

- X is the (possibly infinite) *data set*;
- Φ is a finite set of non-empty (partial) functions of type $X \longrightarrow X$; the set Φ is called the *type* of Z, while each element of Φ called a *processing function* of Z;
- $X_0 \subseteq X$ is the set of *initial data values*;
- Q is the finite set of *states*;
- $q_0 \in Q$ is the *initial state*;
- δ is the (partial) *next-state function*, $\delta : Q \times \Phi \longrightarrow Q$.

An X-machine Z can be regarded as a finite automaton with the arcs labelled by functions from the set Φ; in order to fit precisely Definition 2, the state space can be extended with a non-final "sink" state, that collects all non-defined transitions. The automaton $A_Z = (\Phi, Q \cup \{sink\}, q_0, Q, \delta)$ over the alphabet Φ is called the *associated deterministic finite automaton* (abbreviated *associated DFA*) of Z. Z is said to be *completely defined* if for every $q \in Q$ and every $x \in X$, there exists $\phi \in \Phi$ such that $x \in dom\ \phi$ and $(q, \phi) \in dom\ \delta$. A specification is usually completely defined but the results in this paper are not restricted by this constraint.

Definition 8. A *computation* of Z is a sequence $x_0 \dots x_n$, with $x_0 \in X_0$, $x_i \in X$, $1 \leq i \leq n$, for which there exist $\phi_1, \dots, \phi_n \in \Phi$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n$, and $\phi_1 \dots \phi_n \in L(A_Z)$. The set of all computations of Z is denoted by $Comp(Z)$.

A sequence of processing functions that can be applied in an initial data value is said to be controllable. Controllable sequences are essential in testing as these are associated with the machine computations. An X-machine for which all sequences in the language accepted by the associated DFA are controllable is called itself controllable. As it will transpire in section 6, the X-machine models of spiking neural P systems produced by the method presented in this paper will be controllable and, therefore, suitable for testing purposes.

Definition 9. A sequence $\phi_1 \dots \phi_n \in \Phi^*$, with $\phi_i \in \Phi$, $1 \leq i \leq n$, is said to be *controllable* if there exist $x_0 \in X_0$, $x_1, \dots, x_n \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n$. A set $P \subseteq \Phi^*$ is called controllable if for every $p \in P$, p is controllable. Z is said to be controllable if for every $p \in L(A_Z)$, p is controllable.

Consider now black-box test generation from an X-machine model. In black-box testing, the implementation under test is unknown, but we may assume that it can be modelled by some element from a known fault model. Naturally, when the model is an X-machine Z, we can safely assume that the fault model is a set of X-machines with the same data set X, type Φ and initial data values X_0 as the specification. The underlying strategy for X-machine based testing is to reduce checking that the implementation under test Z' conforms to the model Z to checking that $A_{Z'}$ conforms A_Z and then to apply the W-method. As the W-method, when applied to the associated DFA of the X-machine model, will produce sequences of processing functions, a mechanism for translating sequences of processing functions into sequences of actual data values (used in testing) is needed. Such a mechanism is the *test transformation* defined next.

Definition 10. Given an X-machine $Z = (X, \Phi, x_0, Q, q_0, \delta)$, a *test transformation* of Z is a function $\tau : \Phi^* \longrightarrow X^* \cup \{\perp\}$, $\perp \notin X^*$, such that, for every $\phi_1, \dots, \phi_n \in \Phi$, $n \geq 0$, and $p = \phi_1 \dots \phi_n$, $\tau(p)$ meets the following requirements:

- If p is controllable then
  - If $\phi_1 \dots \phi_n \in L(A_Z)$ then $\tau(\phi_1 \dots \phi_n) = x_0 \dots x_n$ for some $x_0 \in X_0$, $x_1, \dots, x_n \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n$;
  - Else $\tau(\phi_1 \dots \phi_n) = x_0 \dots x_{n_0+1}$, for some $x_0 \in X_0$, $x_1, \dots, x_{n_0} \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n_0$, where n_0 is such that $\phi_1 \dots \phi_{n_0} \in L(A_Z)$ and $\phi_1 \dots \phi_{n_0+1} \notin L(A_Z)$;
- Else $\tau(p) = \perp$.

In other words, τ maps a controllable sequence of processing functions $p = \phi_1 \dots \phi_n \in \Phi$ that is a path in the associated automaton A_Z onto a sequence of data values $x_0 \dots x_n$ that exercises that path. On the other hand, if p is not a path of A_Z, then the longest prefix $p_0 = \phi_1 \dots \phi_{n_0}$ of p that is a path of A_Z is sought and $\tau(p)$ produces a sequence $x_0 \dots x_{n_0+1}$ that attempts to exercise p_0 plus one extra arc, ϕ_{n_0+1}. Given a controllable sequence of processing functions p, $\tau(p)$ will be used in testing to establish if the path p is accepted by the associated automaton of the unknown model Z' of the implementation. On the other hand, non-controllable paths are not useful in testing, so $\tau(p) = \perp$ for any non-controllable sequence p.

As our testing strategy is to reduce checking that the implementation model Z' conforms to Z to checking that $A_{Z'}$ conforms to A_Z, it must be possible to identify the processing functions applied from the computations of Z and Z' examined in the testing process. Therefore, the concept of identifiable type, defined below, is needed.

Definition 11. Φ is called *identifiable* if, for every $\phi_1, \phi_2 \in \Phi$ for which there exists $x \in X$ such that $\phi_1(x) = \phi_2(x)$, then $\phi_1 = \phi_2$.

When Φ is identifiable, it is possible to establish if a controllable sequence of processing functions is correctly implemented by examining the computations of the specification Z and of the implementation Z'. This result is given next.

Theorem 2. *Let $Z = (X, \Phi, x_0, Q, q_0, \delta)$ and $Z' = (X, \Phi, x_0, Q', q_0', \delta')$ be two X-machines and τ a test transformation of Z. If Φ is identifiable then, for*

every $\phi_1, \ldots, \phi_n \in \Phi$ *such that* $p = \phi_1 \ldots \phi_n$ *is controllable, the following holds: if* $\tau(p) \in Comp(Z) \Leftrightarrow \tau(p) \in Comp(Z')$*, then* $p \in L(A_Z) \Leftrightarrow p \in L(A_{Z'})$.

Proof. "$\Longrightarrow$": Suppose $p \in L(A_Z)$. Since p is controllable, by Definition 10, $\tau(\phi_1 \ldots \phi_n) = x_0 \ldots x_n$ for some $x_0 \in X_0$, $x_1, \ldots, x_n \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n$. Then $x_0 \ldots x_n$ is a computation of Z and so $x_0 \ldots x_n$ is a computation of Z'. Then there exist $\phi'_1, \ldots, \phi'_n \in \Phi$ such that $\phi'_1 \ldots \phi'_n \in L(A_{Z'})$ and $\phi'_i(x_{i-1}) = x_i$, $1 \leq i \leq n$. As Φ is identifiable, using a simple induction, it follows that $\phi'_i = \phi_i$, $1 \leq i \leq n$. Hence $p \in L(A_{Z'})$.

"$\Longleftarrow$": Suppose $p \notin L(A_Z)$. Then, by Definition 10, $\tau(\phi_1 \ldots \phi_n) = x_0 \ldots x_{n_0+1}$, for some $x_0 \in X_0, x_1, \ldots, x_{n_0} \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n_0$, where n_0 is such that $\phi_1 \ldots \phi_{n_0} \in L(A_Z)$ and $\phi_1 \ldots \phi_{n_0+1} \notin L(A_Z)$. We provide a proof by contradiction that $\phi_1 \ldots \phi_{n_0+1} \notin L(A_{Z'})$. Assume $\phi_1 \ldots \phi_{n_0+1} \in L(A_{Z'})$. Then $x_1 \ldots x_{n_0+1}$ is a computation of Z' and so $x_1 \ldots x_{n_0+1}$ is a computation of Z. Then there exist $\phi'_1, \ldots, \phi'_{n_0+1} \in \Phi$ such that $\phi'_1 \ldots \phi'_{n_0+1} \in L(A_Z)$ and $\phi'_i(x_{i-1}) = x_i$, $1 \leq i \leq n_0 + 1$. Since Φ is identifiable, by induction it follows that $\phi'_i = \phi_i$, $1 \leq i \leq n_0 + 1$. Then $\phi_1 \ldots \phi_{n_0+1} \in L(A_Z)$. This provides a contradiction, as required. □

The above result can be used to generate a test set from an X-machine model Z in which all paths of the associated automaton are controllable. When only computations whose length does not exceed an upper bound l are of interest, the W-method for bounded sequences (Theorem 1) is applied to produce sequences of processing functions from the associated automaton A_Z, which are then translated into actual data sequences using a test transformation. This is the strategy used in the next section for generating test sequence for an SN P system.

6. A testing approach for SN P systems

We can now use the previously presented results to devise our testing strategy for SN P system models. Essentially, our approach involves two steps:

- Rewriting the SN P system as an X-machine; more specifically, an *approximative* X-machine model Z_l of the system under test will be produced using the L^l algorithm;

- Test suites are derived from Z_l using the W-method for bounded sequences (more precisely, its extension to the X-machines presented here).

First, let us examine how an SN P system can be rewritten as an X-machine. Let $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$, $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, be an SN P system. We consider that Π reads from the environment a binary sequence of 0s and 1s (for convenience, in our running examples we also use a special end marker, but this does not alter the validity of the theoretical results given below). In each step, the first element is removed from the sequence until all elements have been exhausted; if this is an 1, the input neuron will receive a spike.

Then Π can be transformed into an X-machine $Z = (X, \Phi, x_0, Q, q_0, \delta)$, where

- $X = \mathbb{N}^m \times \{0,1\}^*$;

- $\Phi = \{\phi \mid \phi = (\psi_1, \ldots, \psi_m)$ and $\psi_i = \psi_{r_i}$ for some $r_i \in R_i$ or $\psi_i = e_i$, $1 \leq i \leq m\}$ with $e_i \notin \{\psi_{r_i} \mid r_i \in R_i\}, 1 \leq i \leq m$;

- $X_0 = \{(n_1, \ldots, n_m)\} \times \{0,1\}^*$;

- $Q = \{q_0, q_1\}$;

- $\delta(q_0, \phi) = q_0$, $\forall \phi \in \Phi \setminus \{(e_1, \ldots, e_m)\}$; $\delta(q, (e_1, \ldots, e_m)) = q_1$, $\forall q \in Q$.

A data value $x \in X$ will be a tuple holding the configuration of Π and the current binary sequence to be read. An initial data value $x_0 \in X_0$ will hold the initial configuration and the binary sequence originally supplied to Π. The type Φ consists of all processing functions $\phi = (\psi_1, \ldots, \psi_m)$, where ψ_i is either of the form ψ_{r_i}, $r_i \in R_i$, or is e_i, $1 \leq i \leq m$. The component function ψ_{r_i} corresponds to the application of the rule $r_i \in R_i$ in neuron σ_i, $1 \leq i \leq m$, on the configuration of Π, while e_i indicates that no rule is applicable in σ_i; also,

the first element of the binary sequence is removed (passed on as a spike to the input neuron), if this is not empty. While there is at least a neuron in which a rule can be applied, the X-machine Z remains in state q_0; otherwise (when a halting configuration of Π is reached), Z enters state q_1. Consequently, all computations of Π are realized in state q_0 of Z, the sole purpose of state q_1 being to "collect" the non-defined transitions so that Z is completely defined. Finally, note that, although the SN P system might have a final neuron, the corresponding X machine does not consider it as being a special neuron. As mentioned earlier, the values of the spikes in all neurons will be observed.

From the above definition of the X-machine Z associated with the SN P system Π, one can observe that whenever in each computation step, whereby in each neuron σ_i at most one rule r_i is applied, with $r_i \in R_i$, $1 \leq i \leq m$, a unique function $\phi = (\psi_1, \ldots, \psi_m)$, $\phi \in \Phi$ is applied in Z. The component function $\psi_i, 1 \leq i \leq m$, is either ψ_{r_i}, when r_i is applied in σ_i, or is e_i, otherwise.

Example 1. Let us consider the SN P system

$$\Pi_1 = (O, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, syn, 1, 6),$$

where

- $O = \{a\}$;
- $\sigma_1 = (0, \{r_{1,1} : a \to a, r_{1,2} : a^2 \to a^2\})$,
  $\sigma_2 = (1, \{r_2 : a \to a\})$,
  $\sigma_3 = (0, \{r_{3,1} : a \to a, r_{3,2} : a^2 \to \epsilon, r_{3,3} : a^3 \to \epsilon\})$,
  $\sigma_4 = (0, \{r_{4,1} : a^2 \to a, r_{4,2} : a \to \epsilon, r_{4,3} : a^3 \to \epsilon\})$,
  $\sigma_5 = (0, \{r_5 : a \to a\})$,
  $\sigma_6 = (0, \emptyset)$;
- $syn = \{(1,3), (1,4), (2,5), (5,3), (5,4), (3,5), (4,5), (4,6))$.

The diagram of the SN P system Π_1 showing the neuron cells and synapses is presented in Figure 2.

Π_1 reads from the environment a binary sequence such that, for every 1 a spike will be brought into σ_1, whereas a 0 will not bring anything. Furthermore, the sequence ends with a marker, denoted by θ, which brings two spikes into the input neuron. The initial configuration of Π_1 is $(0, 1, 0, 0, 0, 0)$. Π_1 will start by acting on the first element of the input sequence and by sending a spike from σ_2 to σ_5. For each next step until the last element of the input sequence is processed, a spike from σ_5 will be sent to both σ_3 and σ_4 and if a spike is in σ_1 then this will be sent to σ_3 and σ_4 using $r_{1,1}$, otherwise nothing will be sent to them. In the case of a spike received by both σ_3 and σ_4, we end up with two spikes in each of them and in σ_3, $r_{3,2}$ is used and the spikes are removed, whereas in σ_4, $r_{4,1}$ will send a spike to both σ_5 and σ_6. When no spike is received by σ_3 and σ_4 (this means a 0 was read from the environment), we have only one spike in each of the neurons σ_3 and σ_4. Then the rules $r_{3,1}$ and $r_{4,2}$ are applied. One can notice that in each step, σ_5 will receive a spike either from σ_3 or σ_4 and whenever a 1 is read from the environment, a new spike is finally added to σ_6, and when a 0 is read, it will be eventually consumed by σ_3. The whole process stops when the end marker θ is received and two spikes are brought into σ_1; in this case, the rule $r_{1,2}$ will send them to σ_3 and σ_4. Then, with the spike from σ_5, there will be three spikes in each of them and the rules $r_{3,3}$ and $r_{4,3}$ will remove them and the process stops. When the input sequence is not empty, i.e., it has a non empty sequence of 0 and 1, the neuron σ_5 has one more spike received in the previous step, either from σ_3 or σ_4. In the next step, this will be sent to σ_3, which will consume it through $r_{3,1}$ and σ_4, which will remove it, by applying $r_{4,2}$. Hence the final result, obtained in σ_6, will remain unchanged. Therefore Π_1 counts the number of 1s in the input sequence.

Note that one can count 0s as well, simply by introducing a new neuron connected only to σ_3. In this case the rule $r_{3,1}$ will also send a spike to the newly created neuron whenever a single spike appears in σ_3. So, this new neuron will get the number of spikes that gives the number of 0s from the input sequence. In this case, when the input is not empty the number of spikes coming from σ_3 is the number of 0s from the input sequence plus 1.

One can now built the X-machine Z_1 according to the above-mentioned procedure. The initial set of values is $X_0 = \{(0,1,0,0,0,0)\} \times \{0,1\}^*$ (i.e., initially, there will be a spike in σ_2 and nothing in the rest). We now define the functions of Φ as tuples.

The function that will be applied first to a value in X_0 is $\phi_1 = (e_1, \psi_{r_2}, e_3, e_4, e_5, e_6)$. This corresponds to the first step in Π_1, whereby only r_2 is applicable in σ_2, denoted by the function ψ_{r_2} The next functions applicable are $\phi_{2,i} = (\psi_1, e_2, e_3, e_4, \psi_{r_5}, e_6), i = 1, 2$, where $\psi_1 = e_1$, in $\phi_{2,1}$ when there is no spike in σ_1 and $\psi_1 = \psi_{r_{1,1}}$, in $\phi_{2,2}$ when a spike is in σ_1. Then, if the input from the environment has more than one digit, then, in the next steps of the computation, one of the following 4 functions is applicable: $\phi_{3,i} = (\psi_1, e_2, \psi_3, \psi_4, \psi_{r_5}, e_6), 1 \leq i \leq 4$, where $\psi_1 = e_1$, and $\psi_3 = \psi_{r_{3,1}}, \psi_4 = \psi_{r_{4,2}}$, in $\phi_{3,1}$ and $\psi_3 = \psi_{r_{3,2}}, \phi_4 = \psi_{r_{4,1}}$, in $\phi_{3,2}$. Similarly, for $\psi_1 = \psi_{r_{1,1}}$, we define $\phi_{3,3} = (\psi_{r_{1,1}}, e_2, \psi_{r_{3,1}}, \psi_{r_{4,2}}, \psi_{r_5}, e_6)$ and $\phi_{3,4} = (\psi_{r_{1,1}}, e_2, \psi_{r_{3,2}}, \psi_{r_{4,1}}, \psi_{r_5}, e_6)$. Finally, when the end of the input is detected, the following functions are applicable: $\phi_{4,i} = (\psi_{r_{1,2}}, e_2, \psi_3, \psi_4, \psi_{r_5}, e_6)$, $1 \leq i \leq 3$, where $\psi_3 = e_3, \psi_4 = e_4$, when $i = 1$, $\psi_3 = \psi_{r_{3,1}}, \psi_4 = \psi_{r_{4,2}}$, when $i = 2$ and $\psi_3 = \psi_{r_{3,2}}, \psi_4 = \psi_{r_{4,1}}$, when $i = 3$. The last functions are $\phi_{5,1} = (e_1, e_2, \psi_{r_{3,3}}, \psi_{r_{4,3}}, e_5, e_6)$ and $\phi_{5,2} = (e_1, e_2, \psi_{r_{3,3}}, \psi_{r_{4,3}}, \psi_{r_5}, e_6)$. We do not consider the final two steps mentioned for Π_1, when the input is not empty. An extra processing function $\phi_6 = (e_1, e_2, e_3, e_4, e_5, e_6)$ will also be needed for the case in which no rule is applicable in Π_1.

Theorem 3. *Let Π and Z as above and let $c_0, \ldots, c_n \in N^m$, $c_0 = (n_1, \ldots, n_m)$ and $b_1, \ldots, b_k \in \{0,1\}$, $k \geq 1$. Then $c_0 \ldots c_n$ is a computation of Π if and only if $x_0 \ldots x_n$ is a computation of Z that keeps Z in state q_0, where $x_0 = (c_0, b_1 \ldots b_k), \ldots, x_k = (c_k, \epsilon), \ldots, x_n = (c_n, \epsilon)$, if $k \leq n$ and $x_0 = (c_0, b_1 \ldots b_k), \ldots, x_n = (c_n, b_{k-n} \ldots b_k)$, otherwise.*

Proof. Follows from the construction of Z. □

However, the X-machine Z above defined cannot directly be used for testing purposes since it is not controllable (for instance, in our running example

sequences $\phi_1\phi_{3,i}$, $1 \leq i \leq 4$, are not controllable since σ_2 cannot send spikes to σ_3 or σ_4 and, consequently, $\phi_{3,i}$ cannot be applied after ϕ_1).

Our approach is to construct a controllable X-machine model using a learning algorithm. Furthermore, in order to keep state explosion under control, the L^l learning algorithm is used to construct an approximation of the SN P system that preserves all its computations that do not exceed a chosen upper bound.

Let Π be as above and let $l > 0$ be the chosen upper bound. The application of the L^l algorithm for learning an approximative X-machine model Z_l of an SN P system Π is detailed in what follows. The data set X, the set of initial data values X_0 and the type Φ are defined as above; we assume that Φ is identifiable (the implications of this assumption are discussed in more detail in the next section). The L^l algorithm is then used to learn the associated automaton A_{Z_l} of Z_l. However, since the labels Φ are processing functions, not mere symbols, the value of the mapping T will be established by examining the data values produced by the machine.

Consider a sequence of processing functions, $p = \phi_1 \dots \phi_n \in \Phi^*$. Naturally, only sequences whose length does not exceed the upper bound will be of interest, the others will be assigned the value -1. When $n \leq l$, we will distinguish between controllable and non-controllable paths; as, for testing purposes, our model will need to contain only controllable paths, the noncontrollable sequences of processing functions will be excluded by assigning them the value 0. Finally, for controllable paths p, the value of $T(p)$ will be established by examining the data values produced along p. We can now assemble the above situations to provide a definition for T.

Definition 12. Consider the setup for the learning algorithm described above. Then $T : \Phi^* \longrightarrow \{-1, 0, 1\}$ is defined as follows. For a controllable sequence of processing functions $p = \phi_1 \dots \phi_n \in \Phi^n$, $n \geq 0$, we denote by $comp(p)$ any sequence $x_0x_1 \dots x_n$, $x_0 \in X_0$, $x_1, \dots, x_n \in X$ such that $\phi_i(x_{i-1}) = x_i$, $1 \leq i \leq n$. Then, for every $p = \phi_1 \dots \phi_n$, $\phi_1, \dots, \phi_n \in \Phi$, $n \geq 0$, $T(p)$ is defined by:

- If $n \leq l$ then
  - If p is controllable then
    - If $comp(p)$ is a computation of Z then $T(p) = 1$;
    - Else $T(p) = 0$;
  - Else $T(p) = 0$;
- Else $T(p) = -1$.

Naturally, T is not necessarily uniquely defined as, for a controllable p, there may exist many sequences $x_0 x_1 \dots x_n$ as above. On the other hand, it will transpire that the choice of the data values does not influence the outcome of the learning algorithm.

We prove now that Z_l reproduces the behaviour of Z when bounded computations are considered.

Lemma 4. *Let Z, l and Z_l be as above. Suppose Φ is identifiable. Let C_l be the set of all controllable sequences of processing functions in $\Phi[l]$ and $U_l = L(A_Z) \cap C_l$. Then A_{Z_l} is a DFCA of U_l.*

Proof. Let $p = \phi_1 \dots \phi_n \in \Phi^*$ be a controllable sequence of processing functions and let $x_0 \in X_0$, $x_1, \dots, x_n \in X$ as in Definition 12. Since Φ is identifiable, by induction on i, $1 \leq i \leq n$, it follows that $p \in L(A_Z)$ if and only if $x_0 \dots x_n$ is a computation of Z. Then $T(p) = 1$ if and only if $p \in L(A_Z)$. Thus, for any sequence $p = \phi_1 \dots \phi_n \in \Phi[l]$, $T(p) = 1$ if and only if $p \in U$. Therefore the L^l algorithm will return a DFCA of U_l. □

Theorem 5. *Let Z, l and Z_l be as above. Suppose Φ is identifiable. Then $Comp(Z) \cap X[l+1] = Comp(Z_l) \cap X[l+1]$.*

Proof. By Lemma 4, A_{Z_l} is a DFCA of $U_l = L(A_Z) \cap C_l$, with C_l being the set of all controllable sequences of processing functions in $\Phi[l]$. Then the result follows since the non-controllable paths in Z do not produce extra computations. □

Furthermore, from construction, all paths of Z_l whose length does not exceed the upper bound l are controllable.

Theorem 6. *Let $p \in \Phi[l]$. If $p \in L(A_{Z_l})$ then p is controllable.*

Proof. Follows from Definition 12. □

Example 2. Consider again our running example (Π_1 and Z_1 in Example 1). It can be observed that the set of processing functions Φ of Z_1 is identifiable (more precisely, Π_1 is a deterministic SN P system according to Definition 13 given in the next section; Theorem 8 shows that such systems produce X-machines for which Φ is identifiable). Then, according to Lemma 4, for any upper bound l, the learning algorithm presented above will produce a DFCA of U_l, the set of controllable sequences of length up to l in $L(A_{Z_1})$.

Let us examine the sequences of processing functions that can be triggered from some initial data value. Recall that the end of the input string fed to Π_1 (providing two spikes to σ_{in}) is denoted by θ. Then Table 1 shows the sequence of processing functions triggered in Z_1 by sequences of inputs of length up to 4 applied to Π_1.

Table 2 shows the sequences of processing functions of length up to 4 that can be triggered in Z_1; these are the sequences of processing functions from Table 1 and their prefixes of length up to 4.

Using the sequences in Table 2, one can construct a DFCA of U_l for $l \leq 4$. For the sake of simplicity, we only provide a DFCA of U_3, as depicted in Figure 3 (in order to have the model completely defined, a loopback transition labelled by ϕ_6 has been added to q_5).

As all paths of Z_l of length at most l are controllable and Φ is identifiable, we can now apply the W-method for bounded sequences and Theorem 2 to generate test sets from Z_l. The theoretical basis for our testing strategy is Theorem 7 below.

Let $Z_l = (X, \Phi, x_0, Q', q'_0, \delta')$. As discussed in section 2.4, the associated automaton A_{Z_l} can be rewritten as a Mealy machine M_l. Let S be a proper

Input sequence applied	**Functions triggered**
no input	ϕ_6
θ	$\phi_1\ \phi_{4,1}\ \phi_{5,1}$
0θ	$\phi_1\ \phi_{2,1}\ \phi_{4,2}\ \phi_{5,2}$
1θ	$\phi_1\ \phi_{2,2}\ \phi_{4,3}\ \phi_{5,2}$
00θ	$\phi_1\ \phi_{2,1}\ \phi_{3,1}\ \phi_{4,2}\ \phi_{5,2}$
01θ	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{4,3}\ \phi_{5,2}$
10θ	$\phi_1\ \phi_{2,2}, \phi_{3,2}\ \phi_{4,2}\ \phi_{5,2}$
11θ	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{4,3}\ \phi_{5,2}$
000θ	$\phi_1\phi_{2,1}\phi_{3,1}\phi_{3,1}\phi_{4,2}\phi_{5,2}$
001θ	$\phi_1\ \phi_{2,1}\ \phi_{3,1}\ \phi_{3,3}\ \phi_{4,3}\ \phi_{5,2}$
010θ	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{3,2}\ \phi_{4,2}\ \phi_{5,2}$
011θ	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{3,4}\ \phi_{4,3}\ \phi_{5,2}$
100θ	$\phi_1\ \phi_{2,2}\ \phi_{3,2}\ \phi_{3,1}\ \phi_{4,2}\ \phi_{5,2}$
101θ	$\phi_1\ \phi_{2,2}\ \phi_{3,2}\ \phi_{3,3}\ \phi_{4,3}\ \phi_{5,2}$
110θ	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{3,2}\ \phi_{4,2}\ \phi_{5,2}$
111θ	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{3,4}\ \phi_{4,3}\ \phi_{5,2}$

Table 1: Input sequences and sequences of functions triggered

state cover of M_l and W a strong characterisation set of M_l, respectively, and let τ be the test transformation of Z_l. Then, given $k \geq 0$, the set we are after is $T_k = \tau(Y_k)$, where

$$Y_k = S\Phi[k+1](W \cup \{\epsilon\}) \cap \Phi[l] \setminus \{\epsilon\}.$$

Example 3. Consider again our running example and let $l = 3$ and Z_3 the X-machine whose state transition diagram is as represented in Figure 3. States q_0, q_1, q_2, q_3, q_4 and q_5 are reached by $s_0 = \epsilon$, $s_1 = \phi_1$, $s_2 = \phi_1\ \phi_{4,1}$, $s_3 = \phi_1\ \phi_{2,1}$, $s_4 = \phi_1\ \phi_{2,2}$ and $s_5 = \phi_6$, respectively, and these are the sequences of minimum length having this property. Therefore $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$ is a proper state cover of the Mealy machine associated with A_{Z_3}. On the other hand, it

Length	**Sequences triggered**
1	ϕ_6
	ϕ_1
2	$\phi_1\ \phi_{4,1}$
	$\phi_1\ \phi_{2,1}$
	$\phi_1\ \phi_{2,2}$
3	$\phi_1\ \phi_{4,1}\ \phi_{5,1}$
	$\phi_1\ \phi_{2,1}\ \phi_{4,2}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,1}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,3}$
	$\phi_1\ \phi_{2,2}\ \phi_{4,3}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,2}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,4}$
4	$\phi_1\ \phi_{2,1}\ \phi_{4,2}\ \phi_{5,2}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,1}\ \phi_{4,2}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,1}\ \phi_{3,1}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,1}\ \phi_{3,3}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{4,3}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{3,2}$
	$\phi_1\ \phi_{2,1}\ \phi_{3,3}\ \phi_{3,4}$
	$\phi_1\ \phi_{2,2}\ \phi_{4,3}\ \phi_{5,2}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,2}\ \phi_{4,2}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,2}\ \phi_{3,1}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,2}\ \phi_{3,3}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{4,3}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{3,2}$
	$\phi_1\ \phi_{2,2}\ \phi_{3,4}\ \phi_{3,4}$

Table 2: Sequences of functions triggered of length up to 4

can be observed that all pairs of states of A_{Z_3} can be distinguished by singletons; e.g., $w_0 = \phi_1$ distinguishes q_0 from any other state, $w_1 = \phi_{4,1}$ distinguishes q_1 from any other state, $w_2 = \phi_{5,1}$ distinguishes q_2 from any other state, $w_3 = \phi_{4,2}$ distinguishes q_3 from any other state, $w_4 = \phi_{4,3}$ distinguishes q_4 from any other state. Therefore $W = \{w_0, w_1, w_2, w_3, w_4\}$ is a strong characterization set of the Mealy machine associated with A_{Z_3}.

The theorem below shows that, for sequences of length at most l, the test suite constructed above will detect all faults of an implementation whose model is in the fault model consisting of all X-machines whose number of states does not exceed the number of states of Z by more than k.

Theorem 7. *Let Z_l, k and T_k be as above. Suppose Φ is identifiable. Then, for any X-machine $Z'' = (X, \Phi, x_0, Q'', q_0'', \delta'')$ such that $card(Q'') - card(Q') \leq k$, $Comp(Z_l) \cap X[l+1] = Comp(Z'') \cap X[l+1]$ if and only if $Comp(Z_l) \cap T_k = Comp(Z'') \cap T_k$.*

Proof. "$\Longrightarrow$": Since $Comp(Z_l) \cap X[l+1] = Comp(Z'') \cap X[l+1]$, it follows that $Comp(Z_l) \cap X[l+1] \cap T_k = Comp(Z'') \cap X[l+1] \cap T_k$. Since $X[l+1] \cap T_k = T_k$, the required result follows.

"$\Longleftarrow$": Suppose $Comp(Z_l) \cap T_k = Comp(Z'') \cap T_k$.

Let $p \in Y_k$ be a controllable sequence of processing functions. Then $\tau(p) \in T_k$. Since $Comp(Z_l) \cap T_k = Comp(Z'') \cap T_k$, $\tau(p) \in Comp(Z)$ if and only if $\tau(p) \in Comp(Z'')$. Then, by Theorem 2, $p \in L(A_Z)$ if and only if $p \in L(A_{Z''})$. Since $p \in Y_k$ is arbitrarily chosen, it follows that, for every controllable sequence of processing functions $p \in \Phi^*$, $p \in L(A_{Z_l}) \cap Y_k$ if and only if $p \in L(A_{Z''}) \cap Y_k$. Then, by Theorem 1, for every controllable $p \in \Phi^*$, $p \in L(A_{Z_l}) \cap \Phi[l]$ if and only if $p \in L(A_{Z''}) \cap \Phi[l]$. Thus, $Comp(Z_l) \cap X[l+1] = Comp(Z'') \cap X[l+1]$. □

7. Identifiable SNPS

As shown by Theorem 7, the complete fault-detection of our testing strategy is guaranteed if the type Φ of Z is identifiable. In this section we investigate this

property and identify particular cases of SN P systems for which the resulting X-machine Z satisfies this condition.

First, it can be observed that this condition is satisfied whenever the SN P system in question exhibits deterministic behaviour.

Definition 13. An SN P system $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ of degree m is said to be *deterministic* if, for every i, $1 \leq i \leq m$, and every two distinct spiking rules $r_x, r_y \in R_i$ of neuron σ_i of the form $E_x/a^{c_x} \rightarrow a^{p_x}$ and $E_y/a^{c_y} \rightarrow a^{p_y}$, respectively, $L(E_x) \cap L(E_y) = \emptyset$.

It can be observed that the SN P system in Example 1 is deterministic.

Theorem 8. *Let $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ be an SN P system of degree m and $Z = (X, \Phi, x_0, Q, q_0, \delta)$ the resulting X-machine. If Π is deterministic then Φ is identifiable.*

Proof. Let $\phi, \phi' \in \Phi$ be two processing functions of Z, $\phi = (\psi_1, \ldots, \psi_m)$, $\phi' = (\psi'_1, \ldots, \psi'_m)$. Suppose there exists $x \in X$ such that $x \in dom\ \phi \cap dom\ \phi'$. Let $x = (c, b)$, where c is the current configuration of Π and b the binary sequence supplied to Π. Then, for every i, $1 \leq i \leq m$, ψ_i and ψ'_i are applicable in c. Since Π is deterministic, $\psi_i = \psi'_i$, $1 \leq i \leq m$. Thus $\phi = \phi'$. □

A more general case in which the resulting Φ is identifiable is given by the following definition.

Definition 14. An SN P system $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ of degree m is said to be *observable* if for every i, $1 \leq i \leq m$, at least one of the following conditions are met:

1 for every two distinct spiking rules $r_x, r_y \in R_i$ of neuron σ_i of the form $E_x/a^{c_x} \rightarrow a^{p_x}$ and $E_y/a^{c_y} \rightarrow a^{p_y}$, respectively, $L(E_x) \cap L(E_y) = \emptyset$;

2 $(j, i) \notin syn$, for all j, $1 \leq j \leq m$, and for every two distinct spiking rules $r_x, r_y \in R_i$ in neuron σ_i of the form $E_x/a^{c_x} \rightarrow a^{p_x}$ and $E_y/a^{c_y} \rightarrow a^{p_y}$, if $L(E_x) \cap L(E_y) \neq \emptyset$ then $c_x \neq c_y$.

In other words, every neuron of Π either (1) exhibits deterministic behaviour or (2) it cannot receive spikes from other neurons and any of its two spiking rules that overlap remove different number of spikes.

Example 4. Let us consider the following SN P system

$$\Pi_2 = (O, \sigma_1, \sigma_2, syn, 1, 2),$$

where

- $O = \{a\}$;
- $\sigma_1 = (0, \{r_{1,1} : aa/a \to a, r_{1,2} : aa/a^2 \to a, r_{1,3} : a^3 \to \epsilon, r_{1,4} : a^4 \to \epsilon, \})$, $\sigma_2 = (0, \emptyset)$;
- $syn = \{(1, 2)\}$.

The SN P system Π_2 reads from the environment a binary sequence such that, for every 1, a spike will be brought into σ_1, whereas a 0 will not bring anything. The sequence ends with an entity coding for three spikes that will be brought into the input neuron. Whenever two spikes are accumulated in σ_1 either $r_{1,1}$ or $r_{1,2}$ will be used in a non-deterministic manner. The system stops when the final marker is read, by using one of the rules $r_{1,3}, r_{1,4}$. It can be observed that Π_2 is observable but not deterministic.

Theorem 9. *Let $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ be a SN P system of degree m and $Z = (X, \Phi, x_0, Q, q_0, \delta)$ the resulting X-machine. If Π is observable, then Φ is identifiable.*

Proof. Let $\phi, \phi' \in \Phi$ be two processing functions of Z, $\phi = (\psi_1, \ldots, \psi_m)$, $\phi' = (\psi'_1, \ldots, \psi'_m)$. Suppose there exists $x \in X$ such that $x \in dom\ \phi \cap dom\ \phi'$ and $\phi(x) = \phi'(x)$. Let $x = (c, b)$, where c is the current configuration of Π and b the binary sequence supplied to Π. Then, for every i, $1 \leq i \leq m$, ψ_i and ψ'_i are applicable in c. Consider one such i, $1 \leq i \leq m$. If the first condition of Definition 14 is met in neuron i then $\psi_i = \psi'_i$. Otherwise, the second condition must be met. If we assume that $\psi_i \neq \psi'_i$ then the number of spikes in neuron

σ_i after the application of ψ_i will differ from the number of spikes in neuron σ_i after the application of ψ'_i. Consequently $\psi_i(x) \neq \psi'_i(x)$, which contradicts the assumption that $\phi(x) = \phi'(x)$. Thus $\psi_i = \psi'_i$. Hence $\phi = \phi'$. □

8. Discussion

Naturally, the successful application of our method will depend on the ability to construct an X-machine *approximation* suitable for our purposes and the choice for upper bound l is a key aspect in this respect. Obviously, larger l will produce approximations increasingly close to the actual model, but this may result in very complex, often unmanageable models and, on the other hand, very long sequences are not the norm in testing. One possible solution to this problem is the use of coverage criteria for assessing the approximations obtained. Code coverage is a very widespread means of assessing the effectiveness of test suites in white box testing: here the suites are selected so that they cover some selected elements of the program under test. Most usually, these are the program statements (or the nodes in the graph associated with the program) or the branches between nodes [40]. These ideas can be translated to model-based testing, yielding the concepts of state coverage and transition coverage, respectively [4, 40]. An even more powerful coverage criterion is based on the pairs of transition symbols covered by a test suite; the percentage of the pairs of symbols covered by the test suite, also called switch cover, transition-pair or two-trip is considered a powerful test coverage criterion and is included in the British Computer Society standard for software component testing [4]. Using this criterion, we can consider that a DFCA approximation is suitable for testing purposes when, by gradually increasing the upper bound l, this metric stabilizes; previous investigations suggests that this happens for reasonably low values of l [26].

Note also that, when passing from the application of the L^l algorithm for a value of l to the application of the algorithm for the next value, $l + 1$, the DFCA learning procedure does not start anew and the values of S and W from

the previous iteration can be used as starting values for the next iteration of the learning algorithm, with obvious time savings. Also note that the sets S and W produced by L^l are a proper state cover and a strong characterization set, respectively, of the resulting DFCA, and so they can be used in the construction of the test suite after a minimization process (typically S and W contain extra sequences and these will be removed in order to obtain smaller test suites).

9. Conclusions

This paper proposes a testing approach for SN P systems that, under well-defined conditions, ensures that the implementation conforms to the specification and also provides a solution to the state explosion problem through the construction of model approximations. The proposed approach consists of three main steps:

- Rewrite the SN P system in terms of X-machines;
- Construct an approximative X-machine model Z_l of the system using the L^l algorithm for learning DFCA;
- Derive test suites from Z^l using the W-method for bounded sequences (the variant of the method for X-machines).

The paper also investigates the concept of identifiable SN P systems, which is essential for testing these systems. Naturally, a tool for supporting the proposed testing method needs to be provided. This will be the subject of a future paper. Case studies can then be performed to assess the effectiveness of the method and of the strategies for selecting the upper bound proposed in Section 8.

Naturally, one possible future line of research is to seek to extend the current strategy to other classes of P systems (e.g., cell-like P systems). The main problem in this direction could be the maximally parallel mode in which the rules may be applied and, consequently, the construction of the associated X-machine may require some limitations to be imposed on the number of rules applied in each cell in one computation step. Another relevant future research

topic is the development of a direct way of building the DFCA approximations from SN P systems (as well as other P system variants), without the translation of these models into equivalent X-machines.

Acknowledgements. The authors would like to thank the reviewers for their comments that helped improving the quality of the paper.

References

[1] Agrigoroaiei, O., Ciobanu, G., 2010. Reversing computation in membrane systems. J. Log. Algebraic Methods Program. 79 (3–5), 278–288.

[2] Agrigoroaiei, O., Ciobanu, G., 2011. Quantitative causality in membrane systems. In: Proceedings of the 12th International Conference on Membrane Computing. CMC'12. Springer-Verlag, pp. 62–72.

[3] Aman, B., Ciobanu, G., 2020. Reversible computation in nature inspired rule-based systems. J. Membr. Comput. 2 (4), 246–254.

[4] Ammann, P., Offutt, J., 2008. Introduction to Software Testing. Cambridge University Press.
URL https://doi.org/10.1017/CBO9780511809163

[5] Andrei, O., Ciobanu, G., Lucanu, G., 2007. A rewriting logic framework for operational semantics of membrane systems. Theor. Comput. Sci. 373 (3), 163–181.

[6] Angluin, D., 1987. Learning regular sets from queries and counterexamples. Inf. Comput. 75 (2), 87–106.
URL https://doi.org/10.1016/0890-5401(87)90052-6

[7] Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., Tini, S., 2010. Compositional semantics of spiking neural P systems. J. Log. Algebraic Methods Program. 79 (6), 304–316.

[8] Câmpeanu, C., Sântean, N., Yu, S., 1999. Minimal cover-automata for finite languages. In: Revised Papers from the Third International Workshop on Automata Implementation. WIA '98. Springer-Verlag, pp. 43–56.

[9] Câmpeanu, C., Sântean, N., Yu, S., 2001. Minimal cover-automata for finite languages. Theor. Comput. Sci. 267 (1-2), 3–16.

[10] Chow, T. S., 1978. Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. 4 (3), 178–187.

[11] Ciobanu, G., Pinna, G., 2021. Memory associated with membrane systems. J. Membr. Comput. 2 (3), 116–132.

[12] Ciobanu, G., Todoran, E. N., 2017. Denotational semantics of membrane systems by using complete metric spaces. Theor. Comput. Sci. 701, 85–108.

[13] Ciobanu, G., Todoran, E. N., 2018. A semantic investigation of spiking neural P systems. In: Proceedings of the 19th International Conference on Membrane Computing. CMC'19. Springer-Verlag, pp. 108–130.

[14] Díaz-Pernil, D., Peña-Cantillana, F., Gutiérrez-Naranjo, M. A., 2013. A parallel algorithm for skeletonizing images by using spiking neural P systems. Neurocomputing 115, 81–91.
URL https://doi.org/10.1016/j.neucom.2012.12.032

[15] Eilenberg, S., 1974. Automata, Languages, and Machines. Academic Press, Inc.

[16] Gheorghe, M., Ipate, F., 2009. On testing P systems. In: Membrane Computing. Springer-Verlag, pp. 204–216.

[17] Gheorghe, M., Ipate, F., Konur, S., 2016. Testing based on identifiable P systems using cover automata and X-machines. Inf. Sci. 372, 565–578.
URL https://doi.org/10.1016/j.ins.2016.08.028

[18] Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P. J., Lüttgen,

G., Simons, A. J. H., Vilkomir, S. A., Woodward, M. R., Zedan, H., 2009. Using formal specifications to support testing. ACM Comput. Surv. 41 (2), 9:1–9:76.
URL https://doi.org/10.1145/1459352.1459354

[19] Holcombe, M., Ipate, F., 1998. Correct Systems. Building a Business Process Solution. Springer, Applied Computing Series.

[20] Hopcroft, J. E., Motwani, R., Ullman, J. D., 2006. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc.

[21] Ionescu, M., Păun, Gh., Yokomori, T., 2006. Spiking neural P systems. Fundam. Informaticae 71 (2-3), 279–308.
URL http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-08

[22] Ipate, F., 2010. Bounded sequence testing from deterministic finite state machines. Theor. Comput. Sci. 411 (16-18), 1770–1784.

[23] Ipate, F., 2012. Learning finite cover automata from queries. J. Comput. Syst. Sci. 78 (1), 221–244.
URL https://doi.org/10.1016/j.jcss.2011.04.002

[24] Ipate, F., Gheorghe, M., 2009. Finite state based testing of P systems. Natural Computing 8 (4), 833–846.

[25] Ipate, F., Gheorghe, M., 2009. Mutation based testing of P systems. International Journal of Computers, Communication and Control 4 (3), 253–262.

[26] Ipate, F., Stefanescu, A., Dinca, I., 2015. Model learning and test generation using cover automata. Comput. J. 58 (5), 1140–1159.
URL https://doi.org/10.1093/comjnl/bxu032

[27] Körner, H., 2003. On minimizing cover automata for finite languages in O(n log n) time. In: Proceedings of the 7th International Conference on

Implementation and Application of Automata. CIAA'02. Springer-Verlag, pp. 117–127.

[28] Körner, H., 2003. A time and space efficient algorithm for minimizing cover automata for finite languages. Int. J. Found. Comput. Sci. 14 (06), 1071–1086.

[29] Liu, X., Li, Z., Liu, J., Liu, L., Zeng, X., 2015. Implementation of arithmetic operations with time-free spiking neural P systems. IEEE Transactions on NanoBioscience 14 (6), 617–624.

[30] Naranjo, G., Ángel, M., Leporati, A., 2009. Performing arithmetic operations with spiking neural P systems. In: Proceedings of the Seventh Brainstorming Week on Membrane Computing, I. BWMC 2009. pp. 181–198.

[31] Ochirbat, O., Ishdorj, T., Cichon, G., 2020. An error-tolerant serial binary full-adder via a spiking neural P system using HP/LP basic neurons. Journal of Membrane Computing 2 (4), 42–48.

[32] Pagliarini, R., Agrigoroaiei, O., Ciobanu, G., Manca, V., 2012. An analysis of correlative and static causality in P systems. In: Proceedings of the 13th International Conference on Membrane Computing. CMC'13. Springer-Verlag, pp. 323–341.

[33] Păun, A., Păun, Gh., 2007. Small universal spiking neural P systems. Biosyst. 90 (1), 48–60.
URL https://doi.org/10.1016/j.biosystems.2006.06.006

[34] Păun, Gh., 2000. Computing with membranes. Journal of Computer and System Sciences 61 (1), 108–143.

[35] Păun, Gh., 2002. Membrane Computing: An Introduction. Springer-Verlag.

[36] Păun, Gh., Rozenberg, G., Salomaa, A., 2010. The Oxford Handbook of Membrane Computing. Oxford University Press, Inc.

[37] Pinna, G., 2017. Reversing steps in membrane systems computations. In: Proceedings of the 18th International Conference on Membrane Computing. CMC'18. Springer-Verlag, pp. 245–261.

[38] Rong, H., Wu, T., Pan, L., Zhang, G., 2018. Spiking neural P systems theoretical results and applications. In: Enjoying Natural Computing. Springer-Verlag, pp. 258–268.

[39] Rong, H., Yi, K., Zhang, G., Dong, J., Paul, P., Huang, Z., 2013. Automatic implementation of fuzzy reasoning spiking neural P systems for diagnosing faults in complex power systems. Complexity 2019 (4).

[40] Utting, M., Legeard, B., 2007. Practical Model-Based Testing - A Tools Approach. Morgan Kaufmann.
URL http://www.elsevierdirect.com/product.jsp?isbn=9780123725011

[41] Wang, H., Zhou, K., Zhang, G., 2018. Arithmetic operations with spiking neural P systems with rules and weights on synapses. International Journal of Computers, Communications & Control 13 (4), 574–589.

[42] Wang, T., Zhang, G., Zhao, J., He, Z., Wang, J., Pérez-Jiménez, M. J., 2015. Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems. IEEE Transactions on Power Systems 30, 1182–1194.

[43] Zeng, X., Song, T., Zhang, X., Pan, L., 2012. Performing four basic arithmetic operations with spiking neural P systems. IEEE Transactions on NanoBioscience 11 (4), 366–374.

[44] Zhang, G., Rong, H., Neri, F., Pérez-Jiménez, M. J., 2014. An optimization spiking neural P system for approximately solving combinatorial optimization problems. International Journal of Neural Systems 24, 1–16.

[45] Zhang, G., Rong, H., Paul, P., He, Y., Neri, F., Pérez-Jiménez, M. J., 2021. A complete arithmetic calculator constructed from spiking neural P

systems and its application to information fusion. International Journal of Neural Systems 31, 2050055:1–2050055:17.

DFCA learning procedure

Input: S, W and the current observation table T.

Repeat

\— *Check consistency* —\

For every $w \in W$, in increasing order of $length(w) = i$ **do**

Search for $s_1, s_2 \in S$ with $length(s_1), length(s_2) \leq l - i - 1$

and $\sigma \in \Sigma$ such that $s_1 \sim_k s_2$, where

$k = max\{length(s_1), length(s_2)\} + i + 1$, and $T(s_1\sigma w) \neq T(s_2\sigma w)$.

If found **then**

Add σw to W.

Extend T to $(S \cup S\Sigma)W$ using membership queries.

\— *Check closedness* —\

Set $new\_row\_added = false$.

Repeat for every $s \in S$, in increasing order of $length(s)$

Search $\sigma \in \Sigma$ such that $s\sigma \not\sim t \ \forall t \in S$ with $length(t) \leq length(s\sigma)$.

If found **then**

Add $s\sigma$ to S.

Extend T to $(S \cup S\Sigma)W$ using membership queries.

Set $new\_row\_added = true$.

Until new_row_added or all elements of S were processed

Until $\neg new\_row\_added$

Construct $A(S, W, T)$.

Return $A(S, W, T)$.

Figure 1: The learning procedure of L^l

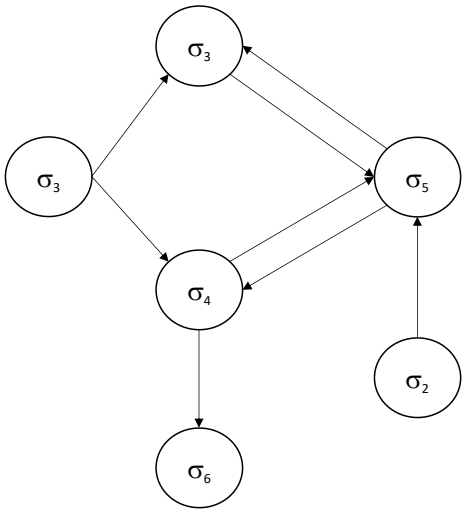

Figure 2: SN P system Π_1 diagram

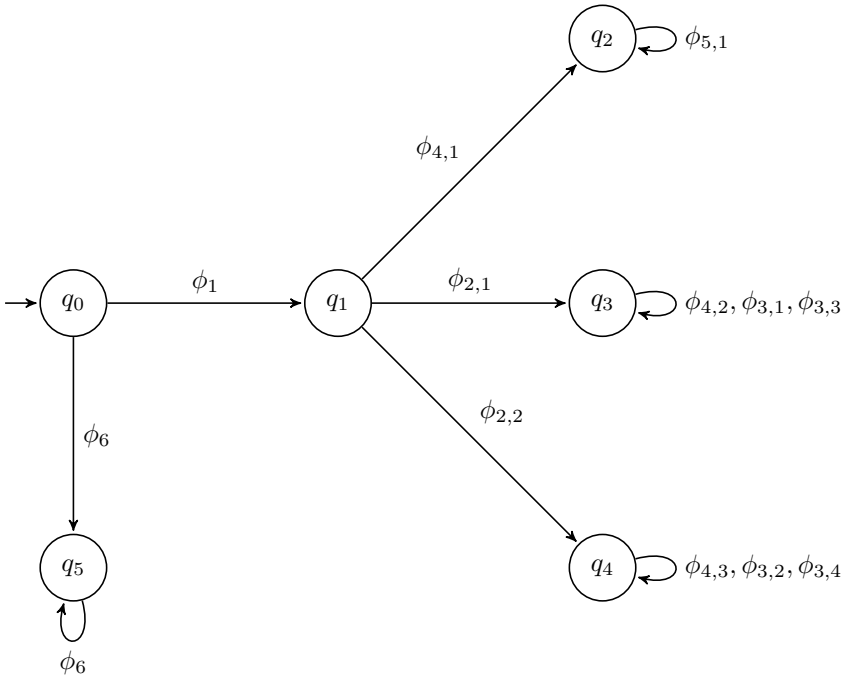

Figure 3: State transition diagram of a DFCA of U_3