



# Spiking neural P systems and their semantics in Haskell

Gabriel Ciobanu<sup>1</sup> · Enea Nicolae Todoran<sup>2</sup>

Accepted: 27 May 2022 / Published online: 10 August 2022  
© The Author(s), under exclusive licence to Springer Nature B.V. 2022

## Abstract

We use the functional programming language Haskell to design semantic interpreters for the spiking neural P systems. Haskell provides an appropriate support for implementing the denotational semantics of a concurrent language inspired by the spiking neural P systems. This language and its semantics describe properly the structure and behaviour of the spiking neural P systems. The semantic interpreters capture accurately the nondeterministic behaviour, the time delays between firings and spikings, and the synchronization specific to spiking neural P systems.

**Keywords** Spiking neural P systems · Semantic interpreter · Denotational semantics · Functional language Haskell

## 1 Introduction

The functional programming language Haskell (<http://haskell.org/>) is used in this paper to describe the spiking neural P systems (Ionescu et al. 2006). This functional implementation is realized by semantic interpreters for a simple concurrent language able to describe the structure and behaviour of spiking neural P systems. The semantic interpreters are designed by using the method of denotational semantics (Schmidt 1986). In fact, we design and implement semantic interpreters for a concurrent language named  $\mathcal{L}_{snp}$  able to describe the structure and behaviour of spiking neural P (shortly, SN P) systems. It provides constructs to describe neurons, synapses and rules with time delays that are specific to SN P systems. Here we emphasize on issues related to implementation, simulation and verification, and provide semantic interpreters describing properly the nondeterministic behaviour, the time delays between firings and spikings, and the synchronized functioning of the SN P systems.

In  $\mathcal{L}_{snp}$ , a spike is represented as an elementary 'statement', and statements can be combined by using an operator for parallel composition. Intuitively, each spike is 'executed' in the context of a neuron. In general, a statement describes a multiset of spikes which are executed concurrently. According to Scott (1980), "it is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct". In order to provide support for automated verification, the nondeterministic systems are modelled in our implementation taking into account all possible interactions in the behaviour of an SN P system. In denotational semantics, such a nondeterministic behaviour is described by using powerdomains; since a specific simulation of an SN P system represents an execution trace, we work with a collections of execution traces implementing elements of a linear time powerdomain (de Bakker and de Vink 1996).

The semantic interpreters of language  $\mathcal{L}_{snp}$  are available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in three variants contained in files `semSNP.hs`, `semSNP-rnd.hs` and `semSNP-fin.hs`. The file `semSNP.hs` implements a semantic interpreter producing all possible execution traces regardless of the length or number of execution traces; in the presence of nondeterminism, this interpreter can only be used to verify toy SN P systems. The file `semSNP-rnd.hs` implements a semantic interpreter which produces a single execution trace and simulates nondeterministic behaviour by using a (pseudo) random number generator. On the other hand, the semantic interpreter implemented in the file `semSNP-fin.hs` is designed to preserve only a finite prefix for each

---

Both authors have contributed equally to this work.

---

✉ Enea Nicolae Todoran  
eneia.todoran@cs.utcluj.ro

Gabriel Ciobanu  
gabriel@info.uaic.ro

<sup>1</sup> Alexandru Ioan Cuza University, Iasi, Romania

<sup>2</sup> Technical University, Cluj-Napoca, Romania

execution trace. Intuitively, each execution trace is stopped after a finite number of steps (no matter whether the system subsequently halts or not). Thus, the interpreter `semSNP-fin.hs` can be used to simulate and verify (bounded) executions of nondeterministic SN P systems.

## 2 SN P systems and their semantics in Haskell

We develop software components able to provide automatic support for simulating and verifying spiking neural P (SN P) systems. For this purpose, we design a simple concurrent language called  $\mathcal{L}_{snp}$  capturing the basic notions specific to an SN P system: neurons, synapses and rules with time delays. We present the syntax of  $\mathcal{L}_{snp}$ , and explain informally the behaviour of its constructs. Recent works describe tools providing support for SN P systems simulation and verification (Gheorghe et al. 2021; Pérez-Hurtado et al. 2022), and formal frameworks for SN P systems (Gheorghe et al. 2021; Verlan et al. 2020; Ciobanu and Todoran 2019, 2022). The language  $\mathcal{L}_{snp}$  is similar to the formalisms described in (Ciobanu and Todoran 2019, 2022), where we presented denotational semantics by using metric spaces in order to describe the behaviour of SN P systems. In this paper, both the syntax and the semantics of language  $\mathcal{L}_{snp}$  are implemented in the purely functional programming language Haskell, a language which provides an appropriate support for a mathematical semantics (such as denotational semantics).

In general, denotational semantics provide a meaning function  $\mathcal{M} : \mathcal{L} \rightarrow \mathbf{D}$ , where  $\mathcal{L}$  is a language and  $\mathbf{D}$  is a mathematical domain used to assign meanings to the language constructs. The main characteristic of such a semantics is its *compositionality*: the semantics of a composed construct is defined solely based on the semantics of its components. Compositionality is a nice property to have because it provides scalability and modularity. In recent articles, we developed denotational semantic models for various kinds of P systems (Ciobanu and Todoran 2017, 2019, 2022). This paper is related to Ciobanu and Todoran (2019, 2022), where we have presented denotational semantics designed with metric spaces (de Bakker and de Vink 1996) for languages inspired by SN P systems similar to the language  $\mathcal{L}_{snp}$  investigated in this work. As in Ciobanu and Todoran (2019) (and unlike Ciobanu and Todoran 2022), we use a specific terminology for programming languages, similar to the terminology employed in de Bakker and de Vink (1996). The semantic model presented in this article is similar to the denotational semantics presented in Ciobanu and Todoran (2022), adapted to the objectives of this paper in which the emphasize is on implementation,

simulation and verification. The denotational semantics given in Ciobanu and Todoran (2022) is designed by using silent steps, needed in the metric setting (de Bakker and de Vink 1996) employed in Ciobanu and Todoran (2022) in order to obtain contractiveness of certain semantic operators. In our Haskell implementation, silent steps would reduce the efficiency of the semantic interpreters (execution traces would be longer). Consequently, in this work we do not use silent steps. It is worth noting that the denotational models given in Ciobanu and Todoran (2019, 2022) are designed by using metric domains (de Bakker and de Vink 1996), which are implemented in this paper as Haskell types. In other words, the denotational semantics is considered here from an engineering viewpoint (Schmidt 1986), emphasizing on implementation, simulation and verification. Rather than using the theory of domains (Gierz et al. 2003) and the theory of metric spaces (de Bakker and de Vink 1996), we use a functional programming language as an implementation tool and as a metalanguage for our denotational semantics.

Comparing the semantics presented in this paper and in Ciobanu and Todoran (2019), the difference is that in Ciobanu and Todoran (2019) the continuations contain all the information regarding declarations and concurrent control aspects, while in this paper we obtain a modular design based on the concepts of *continuation* and *semantic environment*. The information regarding declarations and the execution of rules specific to the SN P systems is contained in semantic environments. Moreover, the concurrent control aspects are described using the concepts of *synchronous continuation* and *asynchronous continuation*, fact which is not articulated in Ciobanu and Todoran (2019). A simulator for SN P systems without delays implemented by using GPUs (Graphics Processing Units) is presented in Cabarle et al. (2011); here we present SN P systems allowing delays between firings and spikings, suggesting the denotational semantics as a sound support for simulation and verification of SN P systems. Thus, we develop software components able to provide automatic support for SN P systems simulation and verification.

A preliminary version of this paper was presented in Ciobanu and Todoran (2021). The possibility of using denotational semantics for SN P systems simulation in random trace semantics was not explored in Ciobanu and Todoran (2021).

### 2.1 Syntax of the language $\mathcal{L}_{snp}$

We use specific mathematical notations to describe the formal syntax of language  $\mathcal{L}_{snp}$  (Definition 1) and to explain Example 1 and Example 2. We present shortly some mathematical preliminaries (for sets, multisets and

regular languages) as in our previous work (Ciobanu and Todoran 2019). We use the notation  $(a, b \in)S$  to introduce the set  $S$  with typical elements  $a, b$  ranging over  $S$ .  $\mathcal{P}_{fin}(S)$  denotes the powerset of all finite subsets of set  $S$ . We denote by  $L(E)$  the language associated with a regular expression  $E$ . Also, we represent a multiset by enumerating its elements between brackets '[' and ']'. For example, [] is the empty multiset. A multiset is an unordered collection, e.g.,  $[a, a, b] = [a, b, a] = [b, a, a]$  is the multiset containing two occurrences of element  $a$  and one occurrence of element  $b$ . A multiset  $u$  is also written in the form  $u = [a_1^{m_1}, \dots, a_n^{m_n}]$ , where  $m_i$  is the multiplicity (number of occurrences) of element  $a_i$  in multiset  $u$ , for  $i = 1, \dots, n$ . For example,  $[a, a, b] = [a^2, b^1]$ .

For more information on formal languages, see (Rozenberg and Salomaa 1998).

In Definition 1 we present the formal syntax for language  $\mathcal{L}_{snp}$ . The basic components are an alphabet  $(a \in)O$  of *spikes* or *objects*, and a set  $(N \in)Nname$  of *neuron names*. We use the set  $(u \in)U$  of all finite multisets over  $O$ , and the set  $(\pi \in)\Pi = \mathcal{P}_{fin}(Nname)$  whose elements  $\pi$  are finite sets of neuron names. To simplify Definition 1, we use the (semantic) notions of a set  $\pi \in \Pi$  and a multiset  $u \in U$  to represent some (syntactic) components for which the order of elements is irrelevant.<sup>1</sup> In this work, both sets and multisets are implemented as Haskell lists. The elements of language  $\mathcal{L}_{snp}$  are syntactic constructions called *statements* and *programs*, and the terms *execution* and *evaluation* are used to describe their behaviour.

**Definition 1** The syntax of the language  $\mathcal{L}_{snp}$  is presented in Table 1, where  $E$  is a regular expression over  $O$ , and  $\vartheta \geq 0$  ( $\vartheta \in \mathbb{N}$ ).

An element  $rs \in Rs$  is a list of rules. A construction  $E/u \rightarrow x; \vartheta$  is called a *firing rule*. A construction  $u' \rightarrow \lambda$  is called a *forgetting rule*. A list of rules  $rs \in Rs$  is valid only if for any pair of (forgetting and firing) rules  $u' \rightarrow \lambda$  and  $E/u \rightarrow s; \vartheta$  contained in  $rs$  the following condition is satisfied:  $\neg(u' \in L(E))$ . Here  $L(E)$  is the language associated with the regular expression  $E$ , and we use the notation  $u' \in L(E)$  to express that there is a permutation of multiset  $u'$  which is an element of the language  $L(E)$  ( $\neg$  is the logical negation operator). Note that the multiset  $u$  occurring on the left-hand side of a firing or a forgetting must be nonempty, i.e.,  $u \neq []$ . Usually we omit a terminating  $r_e$ , and write a non-empty list of rules  $rs \in Rs$ ,  $rs = r_1, \dots, r_i, r_e$ , as  $rs = r_1, \dots, r_i$ . Also, for a list of declarations

$$D = \text{neuron } N_0 \{ rs_0 \mid \pi_0 \}, \dots,$$

$\text{neuron } N_n \{ rs_n \mid \pi_n \}$  to be valid, the neuron names  $N_0, \dots, N_n$  must be pairwise distinct and the name of the first neuron in the list must be  $N_0$  (the name  $N_0$  is special, it is reserved).

Apart minor differences in notations, the definitions for the set of programs  $\rho \in \mathcal{L}_{snp}$ , the set of declarations  $D \in Decl$  and the set of rules  $rs \in Rs$  remain as in Ciobanu and Todoran (2019). A neuron declaration is a construction  $\text{neuron } N \{ rs \mid \pi \}$ , where  $N \in Nname$  is a neuron name,  $rs \in Rs$  is a list of rules and  $\pi \in \Pi$  is a finite set containing the names of the neurons that are connected by synapses (are adjacent or neighbouring) to neuron with name  $N$ . Intuitively, each statement  $s \in S$  is executed by (or in the context of) a neuron.

A program  $\rho \in \mathcal{L}_{snp}$  is a pair  $\rho = (D, s)$ , where  $D = \text{neuron } N_0 \{ rs_0 \mid \pi_0 \}, \dots, \text{neuron } N_n \{ rs_n \mid \pi_n \}$  is a declaration (a list of neuron declarations), and  $s \in S$  is a statement. The execution of program  $\rho = (D, s)$  starts by executing statement  $s$  in the context of neuron with name  $N_0$  (i.e., statement  $s$  is executed in the context of the neuron whose declaration occurs on the first position in the list of declarations  $D$ ). In all other cases, a statement  $s \in S$  is executed in the context of a neuron  $\text{neuron } N \{ rs \mid \pi \}$  if  $s$  occurs in the right-hand side of a rule occurring in the list  $rs$ .

As in the original SN P systems model (Ionescu et al. 2006), a neuron may be in one of the following two states: *open* or *closed*. A neuron only accepts spikes in the open status. Essentially, *firing* and *forgetting* rules are handled as in Ionescu et al. (2006). An  $\mathcal{L}_{snp}$  firing rule is a construct  $E/u \rightarrow s; \vartheta$ , where  $E$  is a regular expression,  $u \in U$  is a multiset,  $s \in S$  is a statement, and  $\vartheta \in \mathbb{N}$  is a natural number denoting a time interval. A statement  $s$  denotes a multiset of spikes that are executed concurrently. Hence, a firing rule  $E/u \rightarrow s; \vartheta$  is similar to an extended SN P firing rule allowing for multiple types of spikes (Ionescu et al. 2011; Păun et al. 2010).

At any moment, each neuron contains a (possibly empty) multiset of spikes. Let  $\text{neuron } N \{ rs_N \mid \pi_N \}$  be a neuron that contains in its list of rules  $rs_N$  a firing rule  $r = E/u \rightarrow s; \vartheta$  and currently stores a multiset of spikes  $u_N$ , such that  $u_N \in L(E)$  and  $u \subseteq u_N$  (the notation  $u_N \in L(E)$  is described in Definition 1, and the condition  $u \subseteq u_N$  means that  $u$  is a submultiset of  $u_N$ ). In this case, neuron  $N$  can *fire* (execute) the rule  $r$  meaning the following: the multiset  $u$  is consumed (in neuron  $N$  remains the multiset of spikes  $(u_N \setminus u)$ , where  $(u_N \setminus u)$  is the multiset difference between  $u_N$  and  $u$ ) and the execution of statement  $s$  is triggered after exactly  $\vartheta$  time units ( $s$  is suspended for the next  $\vartheta$  time units). As in the original SN P systems, the neuron is closed (meaning that it cannot receive new spikes) in the time interval between firing and spiking; after

<sup>1</sup> These components could be defined in BNF as pure syntactic entities, e.g., as lists.

**Table 1** Syntax of the language  $\mathcal{L}_{snp}$ 

Statements	$s(\in S)$	::=	$a$	(spike)		
			$\text{init } \pi$	(initialization)		
			$\text{send } \pi a$	(send statement)		
			$s \parallel s$	(parallel composition)		
Rules	$rs(\in Rs)$	::=	$r_\epsilon \mid r, rs$			
			$r(\in R)$	::=	$E/u \rightarrow s; \vartheta$	(firing rule)
					$u \rightarrow \lambda$	(forgetting rule)
Neuron Declarations	$D(\in Decl)$	::=	$d \mid d, D$			
			$d(\in NDecl)$	::=	$\text{neuron } N \{ rs \mid \pi \}$	(neuron)
Programs	$\rho(\in \mathcal{L}_{snp})$	::=	$D, s.$			

this time interval elapses, the neuron becomes open (hence able to receive spikes) again. A forgetting rule  $u \rightarrow \lambda$  executed by a neuron  $N$  removes the multiset  $u \in U$  if it is executed in a state where neuron  $N$  contains exactly the multiset  $u$ . Rules and the open/closed status are handled as in the original proposal (Ionescu et al. 2006), assuming a global clock for measuring time, each neuron operating in a nondeterministic sequential manner, with at most one rule applied in each step.

A statement  $s \in S$  may be a spike  $a \in O$ , an initialization statement  $\text{init } \pi$  (with  $\pi \in \Pi$ ), a send statement  $\text{send } \pi a$  (with  $\pi \in \Pi$  and  $a \in O$ ), or a parallel composition  $s_1 \parallel s_2$  of two statements  $s_1$  and  $s_2$ .<sup>2</sup> As in Ciobanu and Todoran (2019, 2022), when a spike  $a \in O$  is executed by a neuron  $\text{neuron } N \{ rs_N \mid \pi_N \}$  the spike  $a$  is transmitted to all open neighbouring neurons with names in the set  $\pi_N$ . A statement  $s \in S$  essentially denotes a multiset of spikes that are executed concurrently.

The constructs  $\text{init } \pi$  and  $\text{send } \pi a$  have an initialization effect with no counterpart in the original SN P systems (Ionescu et al. 2006). The execution of a program  $\rho = (D, s)$ , where  $D = \text{neuron } N_0 \{ rs_0 \mid \pi_0 \}, \dots, \text{neuron } N_n \{ rs_n \mid \pi_n \}$ , with  $D \in Decl$  and  $s \in S$ , starts by executing statement  $s$  in the context of neuron  $N_0$  (by convention, the name  $N_0$  is reserved to be used as the name of the neuron whose declaration occurs on the first position in any list of declarations  $D \in Decl$ ). In the initial state only the neuron with name  $N_0$  is *active*. The neuron  $N_0$  is initialized automatically upon system start up as an *open* neuron containing an empty multiset of spikes. In the initial state all other neurons are *idle*, meaning that they cannot participate in interactions (they cannot send and cannot receive spikes), and in order to participate in interactions

they must be initialized explicitly by using statements  $\text{init } \pi$  and  $\text{send } \pi a$ .

When a statement  $\text{init } \pi$  is executed by a neuron  $\text{neuron } N \{ rs_N \mid \pi_N \}$ , each neighbouring neuron with name in the set  $\pi \cap \pi_N$  (that have not been initialized previously) is initialized as an open neuron containing an empty multiset of spikes; here,  $\pi \cap \pi_N$  is the set theoretic intersection of sets  $\pi$  and  $\pi_N$ . The statement  $\text{init } \pi$  has no effect upon neurons that have been initialized beforehand. If the statement  $\text{init } \pi$  is executed by a neuron  $\text{neuron } N \{ rs_N \mid \pi_N \}$  in a state where all neurons with names in the set  $\pi \cap \pi_N$  have been initialized beforehand, then the statement  $\text{init } \pi$  is inoperative.

The statement  $\text{send } \pi a$  has a similar initialization effect. When a statement  $\text{send } \pi a$  is executed by a neuron  $\text{neuron } N \{ rs_N \mid \pi_N \}$ , each neighbouring neuron with name in the set  $\pi \cap \pi_N$  that have not been initialized previously is initialized as an open neuron containing the multiset of spikes  $[a]$  (multiset  $[a]$  contains exactly one occurrence of spike  $a$ , and nothing else). If the statement  $\text{send } \pi a$  is executed by a neuron  $\text{neuron } N \{ rs_N \mid \pi_N \}$  in a state where all neurons with names in the set  $\pi \cap \pi_N$  have been initialized beforehand, then the statement  $\text{send } \pi a$  has no initialization effect, but it transmits the spike  $a$  to all open neurons with names in the set  $\pi \cap \pi_N$ .  $\text{send } \pi a$  is a send operation with target indication given by  $\pi$  (Păun 2002). Note that, spikes  $a' \in O$  occurring outside the scope of any  $\text{send } \pi a$  statement are transmitted to the open neighbouring neurons, but have no initialization effect.

The execution of an  $\mathcal{L}_{snp}$  program involves an initialization phase with no counterpart in the original SN P systems presented in Ionescu et al. (2006). In the SN P model presented in Ionescu et al. (2006), the initial multiset of objects contained in each neuron is part of the system specification. By contrast, an  $\mathcal{L}_{snp}$  program  $\rho = (D, s)$  starts its execution with only one *active* neuron, namely the neuron with name  $N_0$ , which is automatically initialized upon system start up as an open neuron containing an empty multiset of spikes (by convention,  $N_0$  is

<sup>2</sup> The language  $\mathcal{L}_{snp}$  is similar to that in Ciobanu and Todoran (2019, 2022). The set of statements  $(s \in S)$  is similar to the set of statements employed in Ciobanu and Todoran (2019), but there are also differences. The construct  $\text{init } \pi$  is lacking from Ciobanu and Todoran (2019). A version of the construct  $\text{send } \pi a$  is provided in Ciobanu and Todoran (2019), but it has no initialization effect.

the name of the first neuron occurring in any list of declarations  $D \in Decl$ ). All other neurons are *idle* in the initial state, and must be initialized explicitly (by using statements `init  $\pi$`  and `send  $\pi a$` ) in order to participate to interactions.

When an  $\mathcal{L}_{snp}$  program  $\rho = (D, s)$  starts its execution, the system initialization operation can be accomplished with the aid of statement  $s \in S$ , which is executed by the neuron with name  $N_0$ . In case the neuron with name  $N_0$  is connected (by synapses) to all other neurons, this initialization phase can be accomplished in a single step (one time unit). Apart from this initialization phase, an  $\mathcal{L}_{snp}$  program can describe accurately how an SN P system behaves.

## 2.2 Semantic interpreter for the language $\mathcal{L}_{snp}$

We describe the main components of a semantic interpreter for the language  $\mathcal{L}_{snp}$ . The semantic interpreter is available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in the file `semSNP.hs`.

```
type Spike = String
type U      = [Spike]
type Nname = String
type Pi     = [Nname]
```

The types `Spike` and `Nname` implement the sets  $O$  of spikes and  $Nname$  of neuron names, respectively. The types `U` and `Pi` implement the sets  $U$  of multisets of spikes and  $\Pi$  of (finite) sets of neuron names, respectively. Sets and multisets are implemented as Haskell lists. The syntax of language  $\mathcal{L}_{snp}$  presented in Definition 1 can be implemented in Haskell as follows:

```
data S      = Spike Spike | Init Pi | Send Pi Spike | Par S S
data Rule   = FireR (RegExp Spike) U S Int | ForgetR U
type Rules  = [Rule]
type NDecl  = (Nname, Rules, Pi)
type Decl   = [NDecl]
type Prog   = (Decl, S)
```

The types `S`, `Rule`, `Rules`, `NDecl`, `Decl` and `Prog` implement the classes  $S$  of statements,  $R$  of rules,  $R_s$  of lists of rules,  $NDecl$  of neuron declarations,  $Decl$  of declarations and programs of  $\mathcal{L}_{snp}$ , respectively, introduced in Definition 1.

We omit here the definition of the type `RegExp` which we use to handle regular expressions defined over the alphabet of spikes `Spike`. For manipulating regular expressions we use a predicate `(elemRegExp u re)` which accepts as arguments a multiset `u` of spikes and a regular expression `re` and verifies whether any permutation of multiset `u` is an element of the language associated with regular expression `re`.

The final result of the semantic interpreter of language  $\mathcal{L}_{snp}$  is an element of the type `F`. The elements of type `F`

represent *program answers*. In this section `F` is a synonym for type `P`, which implements a linear time *powerdomain* presented in de Bakker and de Vink (1996). Powerdomains are employed in denotational semantics to describe concurrent and nondeterministic behaviour. Nondeterministic systems are modelled in our Haskell implementation taking into account all possible interactions describing the behaviour of an SN P system.

```
type F      = P
type P      = [Q]
data Q      = Qe | Q Obs Q
data O      = O Nname U
type Obs    = [O]
```

An element of the type `P` is a set (implemented as a list) of sequences of type `Q`. An element of type `Q` is a sequence of observables of type `Obs`. `Qe` is the empty sequence. An *observable* of type `Obs` is a set (implemented as a list) of pairs of the type `(O Nname U)`. We recall that `Nname` implements the set of neuron names, and `U` implements the set of all finite multisets of spikes. We use the type `Obs` to represent the current state of a spiking neural P system.

```
prefix :: Obs -> P -> P
prefix obs p = [ Q obs q | q <- p ]
ned :: P -> P -> P
ned p1 p2 =
  [ q | q <- p1 'setUnion' p2, q /= Qe ] 'setUnion'
  [ Qe | Qe <- p1 'setIntersect' p2 ]
bigned :: [P] -> P
bigned []    = [Qe]
bigned (p:ps) = p 'ned' (bigned ps)
```

The mapping `prefix` implements the prefixing of an observable to a program answer. The mapping `ned` describes nondeterminism as a union of behaviours. Note that `(ned p1 p2)` terminates only if both `p1` and `p2` terminate. The mapping `bigned` implements a nondeterministic choice between several alternatives.

```
fe :: F
fe = [Qe]
prefixf :: Obs -> F -> F
prefixf = prefix
bignedf :: [F] -> F
bignedf = bigned
```

We use `fe = [Qe]` to represent termination or the reach of a halting configuration as a final program answer of type `fe :: F`. In this section `F` is a synonym for type `P`, and mappings `bignedf` and `prefixf` behave the same as mappings `bigned` and `prefix`, respectively. In Sects. 3

and 4, the definition of type  $F$  is modified in order to obtain different representations of nondeterministic behaviour.

We present now a denotational semantic function  $\text{den}$  which describes the behaviour of  $\mathcal{L}_{\text{snp}}$  statements in a compositional manner. In the definition of mapping  $\text{den}$ , we use a set of *actions* implemented in Haskell by the type  $\text{Act}$ .

```
data Act = Act Spike Pi | ActInit Pi | ActSend Spike Pi
type W = [Act]
```

The elements of the type  $\text{Act}$  are used to define the semantics of the elementary statements  $a$ ,  $\text{init } \pi$  and  $\text{send } \pi a$ . The type  $W$  implements a set of *multisets of actions*. The type of the semantic function  $\text{den}$  is

$$\text{den} :: S \rightarrow E \rightarrow \text{Pi} \rightarrow D .$$

The type  $S$  implements the class of  $\mathcal{L}_{\text{snp}}$  statements,  $E$  is a type of *semantic environments* and  $\text{Pi}$  is the type of sets (implemented as lists) of neuron names specified above. An element of type  $D$  is called a *computation*. The types  $E$  and  $D$  are defined below.

$$\text{type } E = \text{Act} \rightarrow D .$$

A semantic environment of the type  $E$  is a function which maps an action of type  $\text{Act}$  to a computation of type  $D$ . In the definition of semantic function  $\text{den}$  we use a semantic environment defined as fixed point of a higher order mapping.

```
type D      = Cont -> W -> F
type Cont  = (C,K)
data C     = Ce | C D
type K     = [(Nname,NState)]
data NState = Open U | Closed U Int U D
```

The type of computations is designed by using the *continuation semantics for concurrency (CSC)* technique (Ciobanu and Todoran 2014; Todoran 2000). A computation of type  $D$  is a function which receives as arguments a *continuation* of type  $\text{Cont}$  and a multiset of actions of type  $W$  and yields a final value of the type  $F$ . In the CSC approach a continuation is a structured configuration of computations which can be evaluated concurrently. In the implementation presented in this paper a continuation of type  $\text{Cont}$  is a pair  $(c, k)$ , where  $c$  is an element of type  $C$  and  $k$  is an element of type  $K$ . An element of type  $C$  is called a *synchronous continuation*. A synchronous continuation is a computation which executes the spikes that are transmitted between neurons in the current step.<sup>3</sup> An

<sup>3</sup> The empty synchronous continuation  $\text{Ce}$  is used to handle steps where no spikes are emitted, e.g., because all neurons are closed and no neuron can move to the open status in the current step.

element of the type  $K$  is called an *asynchronous continuation*. An asynchronous continuation is a list of pairs of type  $[(Nname, NState)]$ . An element of type  $Nname$  is a neuron name. An element of type  $NState$  describes the current *state* of a neuron. A neuron can be in one of the following two states: open or closed. The open and closed states are implemented by using the data constructors  $\text{Open}$  and  $\text{Closed}$ , respectively. The open status of a neuron is implemented using a construct  $(\text{Open } u)$ , where  $u :: U$  is the multiset of spikes currently stored by the neuron. The closed status of a neuron is implemented using a construct  $(\text{Closed } u \ t \ \text{ur } d)$ , where  $u :: U$  is the multiset of spikes currently stored by the neuron,  $t :: \text{Int}$  implements the time interval remaining until the neuron produces spikes (becoming open again),  $\text{ur} :: U$  is the multiset of spikes remaining in the neuron after the time interval  $t$  elapses, and  $d :: D$  is a computation whose execution is triggered after the time interval  $t$  elapses.

The techniques specific to denotational semantics can be conveniently implemented in Haskell. Since Haskell supports lazy evaluation, the fixed point combinator can be defined as follows:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

The definition of the semantic mapping  $\text{den}$  uses a semantic operator for parallel composition defined (in the style of denotational semantics) as the fixed point of a higher order mapping  $\text{hopar}$ .

```
type Op = D -> D -> D
hopar :: Op -> Op
hopar op d1 d2 = \ (c,k) w ->
  bignedf [d1 (lift op (C d2) c,k) w, d2 (lift op (C d1) c,k) w]
par = fix hopar
```

The operator  $\text{lift}$  is used to lift an operator on computations of type  $D$  to an operator on synchronous continuations of type  $C$ . The semantic interpreters available at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> use the following definition:

```
lift :: Op -> C -> C -> C
lift op Ce Ce = Ce
lift op Ce f = f
lift op f Ce = f
lift op (C d1) (C d2) = C (op d1 d2)
```

The denotational semantics  $\text{den}$  is a compositional function defined using the semantic operator for parallel composition  $\text{par}$  as follows:

```

den :: S -> E -> Pi -> D
den (Spike a)   env pi = env (Act a pi)
den (Send pi1 a) env pi = env (ActSend a (pi1 'setIntersect' pi))
den (Init pi1)  env pi = env (ActInit (pi1 'setIntersect' pi))
den (Par s1 s2) env pi = (den s1 env pi) 'par' (den s2 env pi),

```

where `setIntersect` is the Haskell implementation of the set theoretic intersection operation.

Finally, the semantic interpreter is a mapping `denp` which receives as argument a program of the type `Prog` and yields a (final) value of the type `F`.

```

denp :: Prog -> F
denp (decl,s) = den s env0 pi0 (Ce,k0) []
  where env0      = fix (hoenv decl)
        ((n0,rs0,pi0):_) = decl
        k0         = [(n0,Open [])]

```

The semantic interpreter `denp` uses a semantic environment `env0 :: E` which stores information about the neuron declarations and applies in a nondeterministic sequential manner the rules contained in declarations. The semantic environment `env0` is defined as the fixed point of a higher order mapping `hoenv`. The Haskell implementation of the mapping `hoenv` is available online (<http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22>).

We illustrate in Example 1 how to use the semantic interpreter described in this section. Further experiments are provided in Sects. 3 and 4.

**Remark 1** In language  $\mathcal{L}_{snp}$ , neuron  $N_0$  can be used to receive the spikes emitted by the output neuron.<sup>4</sup> This technique is also used in the design of  $\mathcal{L}_{snp}$  programs  $\rho_1$  and  $\rho_3$  presented in Example 1 and Example 2, respectively, where the neuron with name  $N_0$  receives the spikes emitted by the output neuron  $N_3$ . As in Ionescu et al. (2006), we consider that the result of a computation is the number of steps elapsed between the first two consecutive spikes produced by the output neuron (assuming the output neuron spikes at least twice, ignoring whether or not the calculation subsequently halts). This convention is used in the both examples (Example 1 and Example 2) presented in this paper.

**Example 1** We consider an  $\mathcal{L}_{snp}$  program  $\rho_1$  implementing the spiking neural P system  $\Pi_1$  given in Ionescu et al. (2006) (Section 5, Figure 2). Before presenting the  $\mathcal{L}_{snp}$  program  $\rho_1$ , we briefly describe the system  $\Pi_1$  using the notation and terminology employed in Ionescu et al.

(2006). As in Ionescu et al. (2006), we use the following convention: when the language associated with regular expression  $E$  is  $L(E) = \{a^i\}$ , we write a firing rule  $E/a^i \rightarrow a; t$  in the abbreviated form  $a^i \rightarrow a; t$ .

$\Pi_1$  is a tuple  $(\{a\}, \sigma_1, \sigma_2, \sigma_3, \{(1,2), (2,3)\}, 3)$ , where  $\{a\}$  is a singleton set ( $a$  is called a *spike*),  $\sigma_1, \sigma_2, \sigma_3$  are *neurons*,  $\text{syns} = \{(1,2), (2,3)\}$  is the set of *synapses*, and 3 is the index of the output neuron ( $\sigma_3$  in this example). Neuron  $\sigma_1$  is a pair  $\sigma_1 = (2n - 1, \{a^+ / a \rightarrow a; 2\})$ , where component  $2n - 1$  is the initial number of spikes contained in  $\sigma_1$ , and  $\{a^+ / a \rightarrow a; 2\}$  is the set of rules describing the behaviour of neuron  $\sigma_1$ . Neurons  $\sigma_2$  and  $\sigma_3$  behave as follows:  $\sigma_2 = (0, \{a^n \rightarrow a; 1\})$  and  $\sigma_3 = (1, \{a \rightarrow a; 0\})$ . In the initial state, neurons  $\sigma_2$  and  $\sigma_3$  contain 0 and 1 spike, respectively. Each pair  $(i, j) \in \text{syns}$  describes a synapse providing support for transmitting spikes from neuron  $\sigma_i$  to neuron  $\sigma_j$ .

Two neurons (namely  $\sigma_1$  and  $\sigma_3$ ) fire in the first time unit. The output neuron  $\sigma_3$  produces its first spike in step 1. Neuron  $\sigma_1$  remains closed for two time units, and in the next step it produces a spike which is transmitted to neuron  $\sigma_2$ . In  $3n$  time units, neuron  $\sigma_2$  receives  $n$  spikes from neuron  $\sigma_1$ . Next, neuron  $\sigma_2$  will fire in step  $(3n + 1)$  and, after a delay of one time unit, it will send a spike to neuron  $\sigma_3$  in step  $(3n + 2)$ . The output neuron  $\sigma_3$  will release its second spike in step  $(3n + 3)$ . Hence, (using the convention presented in Remark 1) the number computed by system  $\Pi_1$  is  $3n + 2 = (3n + 3) - 1 = 3(n + 1) - 1$ .

Following Ionescu et al. (2006), we write a firing rule of the form  $E/[a^i] \rightarrow s; \vartheta$  with  $L(E) = \{a^i\}$  in the abbreviated form  $[a^i] \rightarrow s; \vartheta$ . Such a rule consumes all  $i$  spikes, when the neuron executing it contains exactly the multiset  $[a^i]$ . We recall that  $[a^i]$  is the multiset containing  $i$  occurrences of the spike  $a$ . For example,  $[a^3] = [a, a, a]$ . Also, we use the notation  $s^i$  to represent  $i$  copies of statement  $s$  executed in parallel:  $s^1 = s$  and  $s^{i+1} = s \parallel s^i$ , for  $s \in S$  and  $i \in \mathbb{N}$ ,  $i > 0$ .

Let  $n \in \mathbb{N}, n > 0$ . The program  $\rho_1$  is given by  $\rho_1 = (D_1, s_1)$ , where the statement  $s_1 \in S$  is  $s_1 = (\text{send } \{N_1\} a)^{2n-1} \parallel \text{init } \{N_2\} \parallel \text{send } \{N_3\} a$ , and the declaration  $D_1 \in \text{Decl}$  is given by

$$\begin{aligned}
D_1 = & \text{neuron } N_0 \{ r_\epsilon \mid \{N_1, N_2, N_3\} \}, \\
& \text{neuron } N_1 \{ a^+ / [a] \rightarrow a; 2 \mid \{N_2\} \}, \\
& \text{neuron } N_2 \{ [a^n] \rightarrow a; 1 \mid \{N_3\} \}, \\
& \text{neuron } N_3 \{ [a] \rightarrow a; 0 \mid \{N_0\} \}.
\end{aligned}$$

After the initialization step, neuron  $N_1$  contains (a multiset with)  $2n - 1$  spikes, neuron  $N_3$  contains 1 spike, neuron  $N_2$  contains 0 spikes, and neuron  $N_0$  contains 0 spikes.  $N_3$  is the output neuron. In our  $\mathcal{L}_{snp}$  program  $\rho_1$ , neurons  $N_1, N_2$  and  $N_3$  implement the neurons  $\sigma_1, \sigma_2$  and  $\sigma_3$ , respectively, presented

<sup>4</sup> In the original SN P systems (Ionescu et al. 2006), this role is played by the *environment*; this notion should not be confused with the notion of a *semantic environment* employed in this paper, which is specific to the denotational semantics (de Bakker and de Vink 1996).

in the description of system  $\Pi_1$  given in Ionescu et al. (2006). Apart from the initialization step (which is specific to  $\mathcal{L}_{snp}$ ), the  $\mathcal{L}_{snp}$  program  $\rho_1$  describes accurately the behaviour of system  $\Pi_1$  presented in Ionescu et al. (2006).

As explained in Remark 1, in our implementation neuron  $N_0$  receives the spikes emitted by the output neuron  $N_3$ , and the result is considered to be the number of steps elapsed between the first two consecutive spikes produced by the output neuron. The number computed by both system  $\Pi_1$  presented in Ionescu et al. (2006) and  $\mathcal{L}_{snp}$  program  $\rho_1$  is  $3n + 2$ .

In the Haskell execution of the program  $\rho_1$  we consider  $n = 2$ .<sup>5</sup> In this case, the number of steps elapsed between the first and the second spike produced by output neuron  $N_3$  is  $8 = 3n + 2$ , which coincides with the result obtained in Ionescu et al. (2006) for the system  $\Pi_1$ .

The semantic interpreter is available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in the file `semSNP.hs`.

The  $\mathcal{L}_{snp}$  program  $\rho_1$  is implemented and stored in variable `rho1 :: Prog`.

In the rest of the paper, we write  $e \Rightarrow v$  to indicate that an expression  $e$  evaluates (reduces) to a value  $v$ . For example, by using the interpreter `semSNP.hs`, one can perform the following experiment (which runs  $\mathcal{L}_{snp}$  program `rho1` by evaluating the expression `(denp rho1)`):

`denp rho1 =>`

```
[[["N0", []], ["N3", ["a"]], ["N1", ["a", "a", "a"]], ["N2", []]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a", "a"]], ["N2", []]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a", "a"]], ["N2", []]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a"]], ["N2", ["a"]]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a"]], ["N2", ["a"]]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a"]], ["N2", ["a"]]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a", "a"]], ["N2", ["a", "a"]]] .
  [["N0", ["a"]], ["N3", []], ["N1", ["a"]], ["N2", ["a", "a"]]] .
  [["N0", ["a"]], ["N3", ["a"]], ["N1", ["a"]], ["N2", []]] .
  [["N0", ["a", "a"]], ["N3", []], ["N1", []], ["N2", ["a"]]]]
```

In this experiment and in the experiment presented in Section 4, the output is a set (implemented as a list) of type  $\mathbb{P}$ , where each element is an execution trace of type  $\mathbb{Q}$ , and each execution trace is a sequence of observables of type  $\text{Obs}$ . Each observable is displayed on a separate line, and the observables that make up an execution trace are displayed in chronological order. It is worth noting that an observable (of type  $\text{Obs}$ ) is a list implementing a *set*, and so the order of elements contained in an observable is not important. In this example, the output

of `(denp rho1)` is a set (of type  $\mathbb{P}$ ) containing a single execution trace, where output neuron  $N_3$  produces spikes (received by neuron  $N_0$ ) in steps 1 and 9. Thus, the result of the computation is  $8 = 9 - 1$ .

The  $\mathcal{L}_{snp}$  program  $\rho_1$  given in Example 1 is deterministic: executing `rho1` with our semantic interpreter `(denp rho1)` yields a set containing a single execution trace. In this paper we also aim to handle SN P systems exhibiting nondeterministic behaviour. In Example 2 we present a nondeterministic SN P system (also taken from Ionescu et al. (2006)) and its implementation as an  $\mathcal{L}_{snp}$  program  $\rho_3$ .

**Example 2** We consider an  $\mathcal{L}_{snp}$  program  $\rho_3$  which implements the spiking neural P system  $\Pi_3$  given in Ionescu et al. (2006) (Section 5, Figure 4). We use the same notations and conventions as in Example 1. In Ionescu et al. (2006),  $\Pi_3$  is described by a tuple  $\Pi_3 = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \{(1, 2), (2, 1), (1, 3), (2, 3)\}, 3)$ . The three neurons  $\sigma_1, \sigma_2$  and  $\sigma_3$  behave as follows:  $\sigma_1 = (2, \{a^2/a \rightarrow a; 0, a \rightarrow \lambda\})$ ,  $\sigma_2 = (1, \{a \rightarrow a; 0, a \rightarrow a; 1\})$  and  $\sigma_3 = (3, \{a^3 \rightarrow a; 0, a \rightarrow a; 1, a^2 \rightarrow \lambda\})$ . In the first time unit, all the neurons fire. In particular, the output neuron  $\sigma_3$  emits the first spike in step 1. In a nondeterministic manner, neuron  $\sigma_2$  chooses between rules  $a \rightarrow a; 0$  and  $a \rightarrow a; 1$ . As long as neuron  $\sigma_2$  chooses rule  $a \rightarrow a; 0$ , neurons  $\sigma_1$  and  $\sigma_2$  transmit each other one spike and (together) they transmit two spikes to neuron  $\sigma_3$ , which (applying its forgetting rule  $a^2 \rightarrow \lambda$ ) eliminates the two spikes

in the next step. Alternatively, if neuron  $\sigma_2$  chooses rule  $a \rightarrow a; 1$ , then it moves to the closed status for one time unit; it does not receive the spike emitted by neuron  $\sigma_1$ , and so neuron  $\sigma_2$  remains empty. In the next step, neuron  $\sigma_3$  fires applying the rule  $a \rightarrow a; 1$ , hence it is closed for one time unit and cannot receive the spike issued by neuron  $\sigma_2$ . The spike issued by neuron  $\sigma_2$  is received by neuron  $\sigma_1$ , but it is removed using the forgetting rule  $a \rightarrow \lambda$ . Thus, all the neurons remain empty. Finally, the output neuron  $\sigma_3$  emits its second spike with a delay of 1 time unit since the moment it fired, by applying the rule  $a \rightarrow a; 1$ . Thus, there are at least 2 time units between the two spikes produced by the output neuron  $\sigma_3$ ; in this way, system  $\Pi_3$  generates

<sup>5</sup> In the Haskell implementation available at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22>, this program  $\rho_1$  is stored in variable `rho1 :: Prog` (in all files `semSNP.hs`, `semSNP-rnd.hs` and `semSNP-fin.hs`).



in a nondeterministic manner all natural numbers greater than 1 (2, 3, 4, 5, ...), as in Ionescu et al. (2006).

The  $\mathcal{L}_{snp}$  program  $\rho_3$  presented below is designed to capture the behaviour of spiking neural P system  $\Pi_3$  described in Ionescu et al. (2006), namely to generate all natural numbers greater than 1 (2, 3, 4, 5, ...) in a nondeterministic manner. The program  $\rho_3$  is given by  $\rho_3 = (D_3, s_3)$ , where the statement  $s_3 \in S$  is

$$s_3 = (\text{send } \{N_1\} a)^2 \parallel \text{send } \{N_2\} a \parallel (\text{send } \{N_3\} a)^3$$

and the declaration  $D_3 \in Decl$  is given by

$$\begin{aligned} D_3 = & \text{neuron } N_0 \{ r_\epsilon \mid \{N_1, N_2, N_3\} \}, \\ & \text{neuron } N_1 \{ a^2/[a] \rightarrow a; 0, [a] \rightarrow \lambda \mid \{N_2, N_3\} \}, \\ & \text{neuron } N_2 \{ [a] \rightarrow a; 0, [a] \rightarrow a; 1 \mid \{N_1, N_3\} \}, \\ & \text{neuron } N_3 \{ [a^3] \rightarrow a; 0, [a] \rightarrow a; 1, [a^2] \rightarrow \lambda \mid \{N_0\} \}. \end{aligned}$$

After the initialization step (described by statement  $s_3$ ) neuron  $N_1$  contains 2 spikes (the multiset  $[a, a]$ ), neuron  $N_2$  contains 1 spike, neuron  $N_3$  contains 3 spikes, and neuron  $N_0$  contains 0 spikes. In our  $\mathcal{L}_{snp}$  program  $\rho_3$ , neurons  $N_1, N_2$  and  $N_3$  implement the neurons  $\sigma_1, \sigma_2$  and  $\sigma_3$ , respectively, presented in the description of system  $\Pi_3$  given in Ionescu et al. (2006).

The  $\mathcal{L}_{snp}$  program  $\rho_3$  presented in Example 2 (which implements the SN P system  $\Pi_3$  given in Ionescu et al. (2006)) is designed to generate the natural numbers greater than 1 (2, 3, 4, 5, ...) in a nondeterministic manner. The semantic interpreter presented in this section (available at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in the file `semSNP.hs`) generates all possible execution traces regardless of the length or number of execution traces; thus, it cannot be used to test the  $\mathcal{L}_{snp}$  program  $\rho_3$  given in Example 2. In the following sections we present two alternative implementation options which provide support for simulating and verifying the behaviour of nondeterministic SN P systems, either by choosing at random an arbitrary execution trace or by pruning all execution traces after a (given) finite number of computing steps. The semantic interpreters presented in Sects. 3 and 4 can be used to test both  $\mathcal{L}_{snp}$  programs  $\rho_1$  (given in Example 1) and  $\rho_3$  (given in Example 2).

The implementation techniques presented in this paper are quite general and can be used to simulate and verify several variants of SN P systems. To illustrate the flexibility of these techniques, in the public repository we offer further examples of SN P systems with structural plasticity Cabarle et al. (2015) and inhibitory rules Peng et al. (2020), examples which can be executed by using similar semantic interpreters (available at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in folder `\other-models`).

### 3 Interpreter based on random choice

In general, the number of alternative execution traces of a nondeterministic system may be large, even infinite. The semantic interpreter presented in Section 2 is designed to produce all possible execution traces (regardless of the length or number of execution traces), hence in the presence of nondeterminism it can only be used to test some toy nondeterministic SN P systems.

Here we present an implementation which produces a *single* execution trace and simulates the nondeterministic behaviour of an SN P system as random choice. We implement random choice by using a random number generator from Haskell's library `System.Random`. We define the type `Rand` of random number generators as a type synonym of `System.Random.StdGen`:

```
type Rand = System.Random.StdGen .
```

We also need a mapping for obtaining the next value from the generator; for this purpose, we use the mapping `System.Random.next`.

In this section we present an interpreter for the language  $\mathcal{L}_{snp}$  which simulates nondeterministic behaviour by using a random number generator. This interpreter is reasonably efficient and can be used to test any  $\mathcal{L}_{snp}$  program. At different executions it can produce different results, but at each execution it only generates a single (randomly selected) trace. By using the type `Rand` defined above, the Haskell interpreter presented in this section can be obtained from the semantic interpreter described in Section 2 with only few modifications.

To model nondeterministic behaviour as random choice we redefine the type `F` (implementing the final yield of our semantic interpreter) as follows:

```
type F = Rand -> (Q, Rand) .
```

We also need to change the definitions of associated operators `prefixf`, `bignedf` and `fe`. In this version of our semantic interpreter, the operator `bignedf` simulates a random choice between a finite set of (nondeterministic) alternatives.

```
prefixf :: Obs -> F -> F
prefixf obs f = \r -> let (q,r') = f r in (Q obs q,r')
bignedf :: [F] -> F
bignedf fs = \r -> let (r',rand') = System.Random.next r
                    in (fs !! (r' `mod` (length fs))) rand'
fe :: F
fe = \r -> (Qe,r)
```

No other changes are required. All the other Haskell definitions remain as in Section 2. However, it is convenient to define a function `testRand` for obtaining a different random trace at every new execution.

program with (`testRand rho3`) several times one may obtain a different (randomly selected) trace at each new execution. This is illustrated in the three experiments presented below.

`testRand rho3 ⇒`

```
[("N0", []), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a"]), ("N3", ["a"]) .
 [("N0", ["a"]), ("N2", []), ("N1", ["a"]), ("N3", ["a"]) .
 [("N0", ["a", "a"]), ("N2", []), ("N1", []), ("N3", [])]
```

`testRand rho3 ⇒`

```
[("N0", []), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a"]), ("N3", ["a"]) .
 [("N0", ["a"]), ("N2", []), ("N1", ["a"]), ("N3", ["a"]) .
 [("N0", ["a", "a"]), ("N2", []), ("N1", []), ("N3", [])]
```

`testRand rho3 ⇒`

```
[("N0", []), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", ["a"]), ("N1", ["a", "a"]), ("N3", ["a", "a"]) .
 [("N0", ["a"]), ("N2", []), ("N1", ["a"]), ("N3", ["a"]) .
 [("N0", ["a", "a"]), ("N2", []), ("N1", []), ("N3", [])]
```

```
testRand :: Prog -> IO()
testRand prg =
  do rand0 <- System.Random.newStdGen
     print (fst (denp prg rand0))
```

The Haskell interpreter described in this section is available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in the file `semSNP-rnd.hs`, where the  $\mathcal{L}_{snp}$  programs  $\rho_1$  and  $\rho_3$  (presented in Example 1 and Example 2, respectively) are implemented and stored in variables `rho1 :: Prog` and `rho3 :: Prog`, respectively. One can run  $\mathcal{L}_{snp}$  program `rho1` by evaluating the expression (`testRand rho1`). However, since the  $\mathcal{L}_{snp}$  program `rho1` is deterministic one always obtains the same result no matter how many times the experiment is repeated.<sup>6</sup>

On the other hand, the  $\mathcal{L}_{snp}$  program `rho3` is nondeterministic. Using the interpreter `semSNP-rnd.hs` to run this

In each of the three experiments presented above the output is an execution trace of type  $\mathcal{Q}$ , and each execution trace is a sequence of observables of type  $\text{Obs}$ . Each observable is displayed on a separate line, and the observables that make up an execution trace are displayed in chronological order.

Each output obtained in the three experiments presented above encodes a different value (number). We recall that in each case the value is encoded as the number of steps elapsed between the first two consecutive spikes produced by the output neuron  $N_3$  (see Remark 1 and Example 2). In each experiment the first spike is emitted by the output neuron  $N_3$  (and received by neuron  $N_0$ ) in step 2 (the first step is used for initialization). The next spike is emitted by the output neuron  $N_3$  (and received by neuron  $N_0$ ) in the above three experiments in steps 5, 4, and 10, respectively. Hence, the following numbers are obtained in the three experiments presented above: 3, 2 and 8, respectively. Since the  $\mathcal{L}_{snp}$  program `rho3` is nondeterministic, if we continue such experiments we obtain a random sequence of results (interpreted as numbers). The semantic interpreter `semSNP-rnd.hs` can only be used to test a  $\mathcal{L}_{snp}$  program

<sup>6</sup> The result coincides with the output obtained in Example 1 if we remove a pair of enclosing parentheses '[' and ']'; the output obtained in Example 1 is a *set* (implemented as a Haskell *list*) of traces, rather than a single execution trace.

by producing a single (randomly selected) execution of a nondeterministic  $\mathcal{L}_{snp}$  program.

#### 4 Interpreter based on finite execution traces

The semantic interpreter presented in Section 2 is designed to produce all possible execution traces regardless of the length or number of execution traces. The final result of the semantic interpreter is an element of a linear time powerdomain (de Bakker and de Vink 1996). In the presence of nondeterminism the implementation solution described in Section 2 only provides support to run and verify the execution of toy  $\mathcal{L}_{snp}$  programs. This is not surprising. An element of a powerdomain is exponential in the length of execution traces, hence a direct implementation can lead to an intractable solution.<sup>7</sup> The interpreter presented in Section 3 produces a single execution trace and simulates nondeterministic behaviour by using a random number generator. This interpreter is tractable, but can be used only for simulation purposes and, in general, provides only limited information regarding the behaviour of nondeterministic SN P systems.

In this section we explore an alternative implementation option. The semantic interpreter presented in this section is designed to prune the final yield of the semantic function preserving only a finite prefix for each execution trace. Intuitively, the interpreter stops each execution trace after a given number of steps (accepted as an argument by the semantic interpreter). This solution does not solve the tractability issue, but it can provide support for verifying (bounded versions of) nondeterministic SN P systems. The support for verification is provided by taking into consideration a finite prefix for all possible executions of a non-deterministic SN P system. The non-deterministic SN P system presented in Example 2 ( $\mathcal{L}_{snp}$  program  $\rho_3$ ) is used below to illustrate this approach.

The Haskell interpreter described in this section (available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in file `semSNP-fin.hs`) can be obtained from the semantic interpreter described in Section 2 with only few simple modifications, which are described below.

The Haskell type `F` implements the final yield of our semantic interpreter. The definition of type `F` changes now to:

```
type F = Int -> P .
```

An element of the final domain `F` is a function of type `Int -> P`, which accepts as argument a number representing the length of the finite prefix that is produced for each execution trace contained in a set of type `P`. We recall that an element of type `P` is a set (implemented as a Haskell list) of execution traces of the type `Q` (the types `P` and `Q` are presented in Section 2).

The definition of the prefixing operator `prefixf` is adapted to prune the final yield of the semantic interpreter preserving only a finite prefix of a given length for each execution trace:

```
prefixf :: Obs -> F -> F
prefixf obs f = \l ->
  if l<=0 then [Qe]
  else [ Q obs q | q <- f (l-1) ]
```

The operators `bignedf` and `fe` are easily adapted to the new definition of type `F`.

```
bignedf :: [F] -> F
bignedf fs = \l -> bigned [ f l | f <- fs ]
fe :: F
fe = \l -> [Qe]
```

No other modifications are required. However, note that the semantic interpreter mapping (`denp rho l`) accepts two arguments: a program `rho` of type `Prog`, and an additional argument `l` of type `Int` representing the number of steps after which the execution is stopped for each trace.

The semantic interpreter described in this section can be used to test both programs  $\rho_1$  and  $\rho_3$  (introduced in the examples given in Section 2).

First, we consider the  $\mathcal{L}_{snp}$  program  $\rho_3$  presented in Example 2, which is nondeterministic. As explained in Example 2, the  $\mathcal{L}_{snp}$  program  $\rho_3$  is designed to generate in a nondeterministic manner all natural numbers greater than 1 (i.e., 2, 3, 4, 5, ...). The semantic interpreter described in this section is available online at <http://ftp.utcluj.ro/pub/users/gc/eneiat/nc22> in the file `semSNP-fin.hs`, where the  $\mathcal{L}_{snp}$  program  $\rho_3$  is implemented and stored in variable `rho3 :: Prog`. In this case, the semantic interpreter `denp` accepts as extra argument a natural number indicating a finite number of steps after which execution is stopped for each alternative trace. Using the interpreter `semSNP-fin.hs` to run this program by using (`denp rho3 7`), the following output is obtained:

<sup>7</sup> An element of a powerdomain is a tree-like structure, or a collection of “traces” essentially equivalent to an unfolding of such a tree.



continue to produce spikes in subsequent steps as well, but in our experiment execution is stopped (all execution traces are pruned) after the first 7 steps.<sup>8</sup> Anyway, this example confirms experimentally that the system  $\Pi_3$  presented in Ionescu et al. (2006) ( $\rho_3$  in our implementation) can generate the sequence of numbers 2, 3, 4, 5, ... (i.e.,  $\mathbb{N}^+ \setminus \{1\}$ , where  $\mathbb{N}^+$  is the set of natural numbers without 0).

The interpreter `semSNP-fin.hs` can also be used to test the  $\mathcal{L}_{snp}$  program  $\rho_1$  presented in Example 1. In file `semSNP-fin.hs`, the program  $\rho_1$  is implemented and stored in variable `rho1 :: Prog` for the case when  $n = 2$ , and the execution terminates after 10 steps. Thus, using the interpreter `semSNP-fin.hs` to run the program with `(denp rho1 10)` (or `(denp rho1 z)` with  $z > 10$ ), it is obtained the same output as in Example 1.

Depending on the purpose pursued, the language  $\mathcal{L}_{snp}$  presented in this paper can be modified in various ways, and can be extended with constructions that could express more concisely the behaviour of some SN P systems. For example, one can extend the class of statements  $S$  introduced in Definition 1 by replacing the statement `send  $\pi a$`  with a more general construction `send  $\pi s$` , where the argument  $s$  is an arbitrary statement  $s \in S$  (rather than an elementary spike  $a \in O$  as it is in Definition 1). Intuitively, the construct `send  $\pi s$`  executes the spikes contained in statement  $s$  with target indication given by  $\pi$ . The class of statements  $S$  for the extended language can be defined by  $s ::= a \mid \text{init } \pi \mid \text{send } \pi s \mid s \parallel s$ , implemented in Haskell as

```
data S = Spike Spike | Init Pi | Send Pi S | Par S S .
```

The semantics of a statement `send  $\pi_1 s$`  (implemented by `Send pi1 s`) can be expressed in Haskell by the following equation (which replaces the second equation given in the definition of function `den :: S -> E -> Pi -> D` presented in Section 2):

```
den (Send pi1 s) env pi = den s env (pi1 'setIntersect' pi) .
```

This equation describes in a compositional manner the behaviour of a construct `(Send pi1 s)`. It expresses that the semantics of a statement `(Send pi1 s)` evaluated with respect to semantic environment `env` and set of neuron names `pi` coincides with the semantics of statement  $s$  evaluated with respect to semantic environment `env` and set of neuron names `(pi1`

`'setIntersect' pi)`, where the mapping `setIntersect` implements the set theoretic intersection operation. These new definitions for the class of statements  $S$  and the function `den` can be used with all three versions (given in Sects. 2, 3, and 4) of the semantic interpreter presented in this paper.

By using the construct `send  $\pi s$` , a parallel composition of several send statements `send  $\pi a_1 \parallel \dots \parallel \text{send } \pi a_n$`  could be written more succinctly in the form `send  $\pi (a_1 \parallel \dots \parallel a_n)$` . The construction `send  $\pi s$`  can be implemented as explained above, and could make it easier to describe some SN P systems (but would not enhance the expressiveness of language  $\mathcal{L}_{snp}$ ).

## 5 Conclusion

We implemented the spiking neural P systems by using the functional programming language Haskell. In this paper we present this semantic interpreter of spiking neural P systems designed by using the discipline of denotational semantics (Schmidt 1986). For such an implementation we used a programming language  $\mathcal{L}_{snp}$  providing constructs able to describe the structure of SN P systems, together with its denotational semantics able to describe properly the behaviour of SN P systems. Additionally, we used fixed point semantics and continuations for concurrency. The semantic interpreter captures accurately the nondeterministic behaviour, the time delays between firings and spikings, and the synchronized functioning specific to spiking neural P systems. The spiking neural P systems implemented by our semantic interpreter might be seen as 'executable mathematics' Rabhi and Lapalme (1999); it provides support for simulating and verifying the behaviour of SN P systems. Nondeterministic systems are modelled taking into account all possible interactions describing the behaviour of an SN P system. In order to obtain a tractable solution, only finite execution traces are verified. Additionally, the semantic interpreter can produce a single execution trace, simulating its nondeterministic behaviour by using a (pseudo) random number generator.

There exist many classes of SN P systems (Cabarle et al. 2015; Peng et al. 2020; Păun et al. 2010; Pan et al. 2017; Song et al. 2021). The implementation techniques presented in this paper could be used for the simulation and verification in finite trace semantics of several classes of SN P systems.

<sup>8</sup> Notice that executing `(den rho3 7)` of the nondeterministic program  $\rho_3$  by using the interpreter `semSNP-fin.hs` requires around 120 seconds on a processor Intel(R) Core(TM) i5-7200U with CPU @ 2.50 GHz, while executing the deterministic program  $\rho_1$  produces its output almost instantly.

## References

- Cabarle FG, Adorna HN, Martínez-del-Amor MA (2011) Simulating spiking neural P systems without delays using GPUs. *Int J Nat Comput Res* 2(2):19–31
- Cabarle FG, Adorna HN, Pérez-Jiménez MJ, Song T (2015) Spiking neural P systems with structural plasticity. *Neural Comput Appl* 26(8):1905–1917
- Ciobanu G, Todoran EN (2014) Continuation semantics for asynchronous concurrency. *Fundam Inform* 131(3–4):373–388
- Ciobanu G, Todoran EN (2017) Denotational semantics of membrane systems by using complete metric spaces. *Theor Comput Sci* 701:85–108
- Ciobanu G, Todoran EN (2019) A semantic investigation of spiking neural P systems. *Lect Notes Comput Sci* 11399:108–130
- Ciobanu G, Todoran EN (2021) Spiking neural P systems and their semantics in Haskell. Presented at the International Conference on Membrane Computing, ICMC
- Ciobanu G, Todoran EN (2022) A process calculus for spiking neural P systems. *Inform Sci.* 604:298–319. <https://doi.org/10.1016/j.ins.2022.03.096>
- de Bakker JW, de Vink EP (1996) Control flow semantics. MIT Press, Cambridge
- Gheorghe M, Lefticaru R, Konur S, Niculescu IM, Adorna HN (2021) Spiking neural P systems: matrix representation and formal verification. *J Membr Comput* 3:133–148
- Gierz G, Hofmann KH, Keimel K, Lawson JD, Mislove M, Scott DS (2003) Continuous lattices and domains. Cambridge University Press, Cambridge
- Ionescu M, Păun G, Yokomori T (2006) Spiking neural P systems. *Fundam Inform* 71:279–308
- Ionescu M, Păun G, Pérez-Jiménez MJ, Rodríguez-Patón A (2011) Spiking neural P systems with several types of spikes. *Int J of Comput Commun Control* 6:647–655
- Păun Gh (2002) Membrane computing. An introduction. Springer, Berlin, Heidelberg
- Păun Gh, Rozenberg G, Salomaa A (eds) (2010) Handbook of Membrane Computing. Oxford University Press, Oxford
- Pan L, Păun Gh, Zhang G, Neri F (2017) Spiking neural P systems with communication on request. *Int J Neural Syst* 27(8):1750042
- Peng H, Li B, Wang J, Song X, Wang T, Valencia-Cabrera L, Pérez-Hurtado I, Riscos-Núñez A, Pérez-Jiménez MJ (2020) Spiking neural P systems with inhibitory rules. *Knowl Based Syst* 188:105064
- Pérez-Hurtado I, Orellana-Martín D, Martínez-del-Amor MA, Valencia-Cabrera L, Riscos-Núñez A (2022) A new P-Lingua toolkit for agile development in membrane computing. *Inform Sci* 587:1–22
- Rabhi F, Lapalme G (1999) Algorithms: a functional programming approach. Addison-Wesley, Boston
- Rozenberg G, Salomaa A (eds) (1998) Handbook of formal languages, vol 3. Springer, Berlin
- Schmidt DA (1986) Denotational semantics: a methodology for language development. Allyn & Bacon, Bacon
- Scott DS (1980) What is denotational semantics? MIT laboratory for computer science distinguished lecture series, MIT Cambridge
- Song X, Valencia-Cabrera L, Peng H, Wang J, Pérez-Jiménez MJ (2021) Spiking neural P systems with delay on synapses. *Int J Neural Syst* 31(1):2050042
- Todoran EN (2000) Metric semantics for synchronous and asynchronous communication: a continuation-based approach. *Electron Notes Theor Comput Sci* 28:101–127
- Verlan S, Freund R, Alhazov A, Ivanov S, Pan L (2020) A formal framework for spiking neural P systems. *J Membr Comput* 2(4):355–368

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.