UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Sistemas P por impulsos neuronales multicanal: formulación de una variante del modelo y aplicación a la codificación mulidimensional.

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: Llanes Lacomba, Rodrigo

Tutor/a: Sempere Luna, José María

CURSO ACADÉMICO: 2022/2023

Contents

1 Introducción

A lo largo de este trabajo presentaremos un nuevo modelo de computación natural, como una nueva variante del modelo SNP-MC System, explicaremos sus propiedades, diferencias con los modelos anteriores y funcionamiento.

Posteriormente implementaremos un simulador para poder probar el modelo y un lenguaje de especificación con su respectivo intérprete para generar los modelos con mayor comodidad.

Y por último obtendremos una serie de resultados teóricos sobre este modelo, obteniendo universalidad, la capacidad de simular otros modelos y la utilidad intrínseca del modelo para la codificación y decodificación multidimensional, así como para el tratamiento de señales multicanal.

2 Conceptos básicos

En esta sección desarrollaremos los conceptos básicos de computación con membranas necesarios para comprender el modelo que posteriormente presentaremos, así como los desarrollos previos en SNP Systems que han allanado el camino a este trabajo.

2.1 Lenguajes formales

Al trabajar con modelos computacionales es inevitable toparse con la teoría de lenguajes formales, por ello introduciremos unos conceptos básicos extraídos del libro de John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman, Introduction to automata theory, languages, and computation[3].

Definimos un alfabeto Σ como un conjunto finito no vacío de símbolos, sobre el que podemos aplicar las mismas operaciones que sobre los conjuntos (unión, intersección, diferencia, etc). Por ejemplo, dados los siguientes alfabetos:

$$\Sigma_1 = \{a, b\} \qquad \Sigma_2 = \{b, c\}$$

podemos realizar las siguientes operaciones sobre ellos:

$$\Sigma_1 \cup \Sigma_2 = \{a, b, c\}$$

$$\Sigma_1 \cap \Sigma_2 = \{b\}$$

$$\Sigma_1 - \Sigma_2 = \{a\}$$

Podemos combinar los símbolos de un alfabeto para obtener cadenas. Una cadena x sobre un alfabeto Σ se define como una secuencia finita y ordenada de símbolos de Σ. Una cadena de Σ_1 podría ser por lo tanto:

$$x_1 = aabb$$

Un lenguaje L es cualquier conjunto de cadenas, no necesariamente finito (incluyendo el conjunto vacío). Por ejemplo, podemos definir un lenguaje L_1 como:

$$L_1 = \{a^n b^n : n > 0\}$$

Además de los operadores típicamente empleados sobre conjuntos, existen otros dos operadores ampliamente utilizados sobre alfabetos, el cierre de Kleene o estrella de Kleene (*) y el operador más o estrella positiva ($^+$). Se define la estrella de Kleene sobre un alfabeto Σ Σ^*, como el conjunto de todas las cadenas posibles creadas a por concatenación de símbolos de Σ (incluida la cadena vacía denotada como ε). Por ejemplo:

$$\Sigma = \{a, b\}$$

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$$

Por otro lado, la estrella positiva sobre un alfabeto Σ, se define a partir de la estrella de Kleene, como el conjunto resultante de eliminar la cadena vacía de Σ^*, por ejemplo:

$$\Sigma = \{a, b\}$$

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, ...\}$$

2.2 Máquina de registros

Una máquina de registros se define como una construcción de la forma:

$$M = (m, H, l_0, l_h, I)$$

donde:

- $m \geq 1$ indica el número de registros de la máquina (todos inicializados a cero al inicio de la ejecución).
- H es un conjunto de etiquetas de instrucción.
- l_0 es la etiqueta correspondiente a la instrucción de inicio.
- l_h es la etiqueta correspondiente a la instrucción de salto o parada.
- I es el conjunto de instrucciones de manera que $|I| = |H|$ y cada etiqueta de H referencia a una instrucción en I. Cada una de las instrucciones de I es de una de las siguientes formas:
  - $l_i : (ADD(r), l_j, l_k)$ incrementa en una unidad el registro r y luego salta de manera indeterminista a l_j o l_k.
  - $l_i : (SUB(r), l_j, l_k)$ si el contenido de r es mayor que cero le resta uno y salta a l_j, en caso contrario salta a l_k.
  - $l_i : HALT$ detiene la ejecución.

La máquina de registros empieza su ejecución en la instrucción etiquetada con l_0, la ejecuta y salta a la siguiente en función del tipo de instrucción y el valor de los registros, este proceso se repite hasta que llega a la instrucción de parada l_h.

2.3 Multiconjuntos

Definimos un multiconjunto como el par (A, m) donde A es un conjunto y $m : A \to \mathbb{N}$ es una función de A a $\mathbb{N}$ (los número naturales), de esta manera añadimos multiplicidad al conjunto A.

Normalmente solemos representar un multiconjunto como cadenas de la forma $s_1^{n_1} s_2^{n_2} ... s_k^{n_k}$ donde s_i es el i-esimo símbolo de A y n_i es la multiplicidad de dicho símbolo para toda $1 \leq i \leq k$, siendo $k = |A|$.

2.4 Sistemas P

Los sistemas P fueron presentados por primera vez en [9] por Gheorghe Păun, siendo estos un modelo de computación natural inspirado en el funcionamiento interno de una célula, con una membrana exterior llamada piel y varios orgánulos internos separados por membranas que intercambian compuestos químicos (objetos) en función de unas reglas. De este modo, la configuración inicial del sistema dictamina como se realizará el computo.

Las dos grandes ventajas de este sistema son su indeterminismo y su paralelismo intrínseco, pues al estar compuesto por unidades independientes que interactúan entre sí la paralelización es inmediata y natural.

Formalmente un sistema P de transición de grado $n \geq 1$ es una construcción de la forma:

$$\Pi = (V, \mu, w_1, ..., w_n, (R_1, p_1), ..., (R_n, p_n), i_o)$$

donde:

a) V es un alfabeto a cuyos símbolos llamaremos objetos.

b) μ es una estructura de membranas de grado n, con cada membrana etiquetada con un valor único $1, 2, ..., n$. Solemos representar esta estructura como una cadena parentizada.

c) $w_i, 1 \leq i \leq n$ son cadenas de V^* representando multiconjuntos sobre V, asociados a la región i de μ.

d) $R_i, 1 \leq i \leq n$ son conjuntos finitos de reglas asociadas a la región i de μ. p_i es una relación de orden parcial sobre R_i que especifica una prioridad entre dichas reglas. Una regla de evolución es un par (u, v), que normalmente expresaremos como $u \to v$, donde u es una cadena sobre V y $v = v'$ o $v = v'\delta$, donde v' es una cadena sobre

$$[(V \times \{here, out\}) \cup (V \times \{in_j \mid 1 \leq j \leq n\})]^*$$

y δ es un símbolo especial $\delta \notin V$.

e) i_o es un número entre 1 y n, que especifica la membrana de salida.

Cabe destacar que cualquier multiconjunto $m(w_1), ..., m(w_n)$ puede estar vacío (es decir, w_i puede ser λ) y lo mismo se aplica a los conjuntos $R_1, ..., R_n$ y sus relaciones de prioridad asociadas p_i .

A partir de esta descripción podemos pasar a analizar su funcionamiento, por simplicidad vamos a asumir máximo paralelismo (aunque existen otras formas de interpretar el funcionamiento de dichos sistemas). Siguiendo el esquema de máximo paralelismo, nos encontramos con que en cada instante de tiempo se ejecutan de manera simultanea todas las reglas posibles en todas las membranas del sistema, si varias reglas compiten por un mismo recurso se ejecutará de manera indeterminista una de ellas el máximo número de veces posibles.

Una regla $u \to v$ es aplicable en la membrana i cuando $u \subseteq w_i$, en cuyo caso consumirá los símbolos u en w_i y enviará los objetos de v su membrana padre si v es de la forma V^*out, a si misma si v es de la forma V^*here o a su membrana hija j si v es de la forma V^*in_j. Además, si v contiene el símbolo δ la membrana i será disuelta y tanto sus objetos como sus membranas hijas serán enviados a su membrana padre.

También es importante notar que los objetos enviados en el instante t no llegarán al destino (y por lo tanto no podrán disparar ninguna regla) hasta el instante $t + 1$.

Cuando no se pueda ejecutar ninguna regla, la ejecución termina y decimos que hemos llegado a un estado de parada. El resultado de la computación, serán los objetos que se encuentren en la membrana de salida i_o al alcanzar el estado de parada.

Para comprender mejor el funcionamiento del sistema, podemos ver el ejemplo de la figura 1.

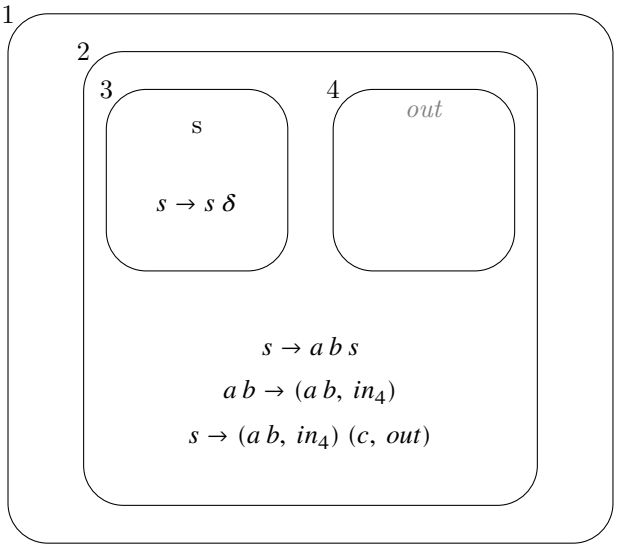

Figure 1: Ejemplo de sistema P

En el primer instante de tiempo, solo se puede aplicar la regla de la membrana 3, por lo que se consume un símbolo s y se genera otro junto con un símbolo de disolución δ, de modo que la membrana 3 se disuelve y todo su contenido cae en la membrana 2.

En el segundo instante de tiempo en la membrana 2 tenemos el símbolo s, por lo que se pueden disparar cualquiera de las dos reglas:

- $s \to abs$, si se activa esta regla, mantendremos s generando a y b, que serán consumidos en el siguiente instante de tiempo por la regla $ab \to (ab, in_4)$ y enviados a la membrana de salida (4).
- $s \to (a\ b,\ in_4)\ (c,\ out)$, Enviamos una a y una b a la membrana de salida y una c a la membrana padre (1). De este modo la primera regla no podrá volver a ejecutarse, impidiendo la generación de nuevos símbolos a y b, y forzando la parada.

Como se puede deducir, el sistema genera los multiconjuntos denotados por el lenguaje $\{a^n b^n : n > 0\}$.

A partir de este modelo se han ido desarrollando gran número de variantes u otros modelos basados en este trabajo, pero vamos a centrarnos en los *SNP Systems* dentro de los cuales se incluye la variante que presentamos en este trabajo.

2.5 Sistemas P con impulsos neuronales

Los sistemas P con impulsos neuronales, Spiking Neural P Systems o SNP Systems, fueron presentados en [4] por Mihai Ionescu, Gheorghe Păun y Takashi Yokomori, como un modelo basado en el funcionamiento de las neuronas. Estas neuronas están conectadas en red y se envían impulsos eléctricos (spikes) en función de unas reglas de activación. El resultado de la computación se obtiene a partir de los spikes que el sistema envía al entorno.

Formalmente podemos describir un SNP System como una construcción de la forma:

$$\Pi = (O, \sigma_1, ..., \sigma_m, syn, out)$$

donde:

a) $O = \{a\}$ es un alfabeto de spikes con un único símbolo.

b) $\sigma_i = (n_i, R_i), 1 \leq i \leq m$ son neuronas, donde:

- $n_i \geq 0$ es el número de spikes que contiene originalmente la neurona.
- R_i es un conjunto finito de reglas de la neurona, de una de las formas:
  - Reglas de impulso:

    $$E/a^c \rightarrow a; d$$

    Donde E es una expresión regular sobre O que debe cumplir el contenido de la neurona para activar la regla, a^c es el número de spikes que consume la regla al activarse para $c > 0$ y d también llamado retardo, es el número de instantes de tiempo que tardará la neurona en ser susceptible de volver a activarse después de lanzar la regla. Tras d instantes de tiempo, la regla emitirá un impulso.
  - Reglas de olvido.

    $$a^s \rightarrow \lambda$$

    Para $s > 0$ la regla consumirá a^s spikes y no producirá activación (no enviará spikes por sus sinapsis). Nótese que al no tener expresión regular pueden ejecutarse siempre que existan en la neurona s o más spikes.

c) $syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ conjunto de sinapsis (conexiones entre neuronas), $(i, i) \notin syn$ para $\forall 1 \leq i \leq m$ pues no existen bucles.

d) $out \in \{1, 2, ..., m\} \cup \infty$ neurona de salida siendo ∞ que la neurona de salida es el entorno.

Su funcionamiento es muy similar al de los sistemas P, con la excepción de que se lanzará como máximo una regla por instante de tiempo en cada neurona, si varias reglas se pueden activar en el mismo instante de tiempo se escogerá de manera indeterminista (con

la excepción de las reglas de olvido que siempre serán las menos prioritarias), cuando una regla $E/a^c \rightarrow a;d$ se ejecuta, se envía una copia de a a todas las neuronas a las que está conectada mediante sinapsis, si $d > 0$ la neurona tardará d instantes de tiempo en emitir el spike, tiempo durante el que la neurona quedará bloqueada y no podrá activar ninguna otra regla ni recibir ningún impulso.

Al igual que con los sistemas P, podemos decir que se alcanza un estado de parada si en un instante de tiempo no se puede aplicar ninguna regla, terminando la ejecución y obteniendo el resultado de la computación como el número de spikes en la neurona de salida. Cabe destacar que existen múltiples formas de leer la salida del sistema, siendo otra variante por ejemplo, los generadores, en los que no hay parada y la salida es el número de spikes en la neurona *out* en cada instante de tiempo.

Para comprender mejor el funcionamiento del modelo, explicaremos más en detalle el funcionamiento del modelo de la figura 2.

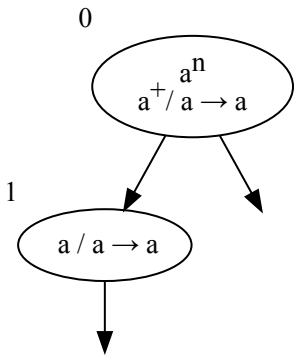

Figure 2: Ejemplo de SNP System

Tenemos un sistema formado por dos neuronas, donde la neurona 0 es la neurona de entrada.

En la primera iteración solo puede aplicarse la regla $a^+/a \rightarrow a$ de la neurona 0, que consumirá un spike y enviará una copia al entorno y otra a la neurona 1.

En la iteración siguiente ambas neuronas se activarán, enviando dos spikes al entorno, este proceso se repetirá $n-1$ veces siendo n el número de spikes de enviados como input al sistema.

En el instante de tiempo $n+1$ en la neurona 0 no quedarán spikes, por lo que solo la neurona 1 activará la regla y enviará un impulso al entorno. En el instante siguiente no habrá ninguna regla aplicable y el sistema alcanzará un estado de parada habiendo generado $n \times 2$ spikes.

Este modelo da el pistoletazo de salida a los SNP Systems y es el causante de que en estos modelos se emplee la terminología de neurona en lugar de membrana (como es costumbre en la mayoría de sistemas P), por este motivo a lo largo del trabajo emplearé ambos términos indistintamente.

En [10] Hong Peng, Jinyu Yang, Jun Wang, Tao Wang, Zhang Sun, Xiaoxiao Song, Xiaohui Luo y Xiangnian Huang presentan la variación en la que nos hemos basado para crear nuestro modelo, los Sistemas P con impulsos neuronales multi-canal o SNP-MC System,

la idea fundamental tras este sistema es añadir múltiples canales de comunicación entre neuronas, de manera que las reglas de activación puedan decidir a través de que sinapsis enviar señal y a través de cuales no.

Formalmente el modelo puede describirse como una construcción de la forma:

$$\Pi = (O, L, \sigma_1, ..., \sigma_m, syn, out)$$

donde:

a) $O = \{a\}$ es un alfabeto de spikes con un único símbolos.

b) $L = \{1, 2, ..., N\}$ es el alfabeto de etiquetas de canal.

c) $\sigma_i = (n_i, L_i, R_i), 1 \leq i \leq m$ son neuronas, donde:

- $n_i \geq 0$ es el número de spikes que contiene originalmente la neurona.
- $L_i \subseteq L$ es un conjunto finito de etiquetas de canal empleadas en σ_i.
- R_i es un conjunto finito de reglas de la neurona, de una de las formas:
  - Reglas de impulso:

    $$E/a^c \rightarrow a^p(l)$$

    Donde E es una expresión regular sobre O que debe cumplir el contenido de la neurona para disparar la regla, a^c es el número de spikes que consume la regla al activarse para $c > 0$ y a^p para $p \geq 0$ y $c \geq p$ es el número de spikes enviados a través del canal $l \in L_i$.

d) $syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\} \times L$ conjunto de sinapsis (conexiones entre neuronas), $(i, i, l) \notin syn$ para $\forall 1 \leq i \leq m$ y para $\forall l \in L$ pues no existen bucles.

e) $out \in \{1, 2, ..., m\}$ indicando la neurona de salida.

Una vez entendemos el funcionamiento de los SNP Systems, los SNP-MC System resultan intuitivos, pues el único añadido es la capacidad de elegir a través de que sinapsis envía los spikes la función de activación, además de permitir enviar más de uno y eliminar el retardo.

Además se incorpora la capacidad de que una neurona ejecute todas las reglas que le sea posible de manera secuencial en el mismo instante de tiempo, manteniendo el paralelismo con el resto de neuronas.

Para terminar de entender el funcionamiento de los SNP-MC Systems, veremos el ejemplo de la figura 3.

En la primera iteración solo puede activarse la neurona σ_1, enviando dos spikes a la neurona σ_2.

En la segunda iteración se escogerá una regla de manera indeterminista, si la regla escogida es la que envía por el canal 1, se enviarán dos spikes al entorno y dos spikes a la neurona σ_1, repitiendo el proceso. Si por el contrario se escoge la regla que envía por el canal 2, solo se enviarán los spikes al entorno y terminará la computación.

Como se puede observar, es un sistema generador del lenguaje $\{a^{n \times 2} : n > 0\}$.

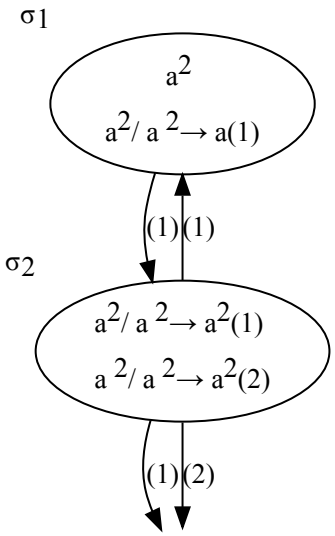

Figure 3: Ejemplo de SNP-MC System

Una vez nos hemos introducido en la computación con membranas y hemos visto los modelos que más han influido en el desarrollo de nuestra variante, podemos pasar a la siguiente sección donde presentaremos dicho modelo.

3 Sistema P con impulsos neuronales multi-canal y múltiples spikes

Tras analizar el ya comentado SNP-MC System, surge de manera intiutiva la idea de añadir nuevos tipos de spikes, si ya estamos trabajando con distintos canales para controlar el uso de las sinapsis. ¿Por que no añadir multiples spikes para simular distintos tipos de impulsos neuronales que accionen distintas reglas de activación?

De esa idea surge nuestra nueva propuesta, recuperando algunas caracterísiticas de los SNP Systems clasicos, combinandolas con los SNP-MC Systems y añadiendo distintos tipos de spikes obtenemos los *Sistema P con impulsos neuronales multi-canal y múltiples spikes.*

Formalmente podemos definir un *sistema P con impulsos neuronales multi-canal y múltiples spikes* como una construcción de la forma:

$$\Pi = (O, L, \sigma_1, ..., \sigma_m, syn, in)$$

donde:

a) O es un alfabeto de spikes con uno o más símbolos.

b) $L = \{1, 2, ..., N\}$ es el alfabeto de etiquetas de canal.

c) $\sigma_i = (w_i, L_i, R_i), 1 \leq i \leq m$ son neuronas, donde:

- $w_i \subseteq O^*$ representando un multiconjunto sobre O, indicando el contenido de la neurona.
- $L_i \subseteq L$ es un conjunto finito de etiquetas de canal empleadas en σ_i.
- R_i es un conjunto finito de reglas de la neurona, de una de las formas:
  - Reglas de impulso:

    $$E/c \rightarrow p_1(l_1), p_2(l_2), ..., p_n(l_n); d$$

    Donde E es una expresión regular sobre O que debe cumplir el contenido de la neurona para disparar la regla, $c \subseteq O^+$ representando el multiconjunto de spikes que consume la regla al activarse para, $p_i \subseteq O^+ \forall 1 \leq i \leq n$ representa el multiconjunto de spikes que enviará la regla a través del canal $l_i \in L_i$ al activarse y d también llamado retardo, es el número de instantes de tiempo que tardará la neurona en generar y enviar la parte derecha, tiempo durante el cual será incapaz de disparar otra regla.
  - Reglas de olvido.

    $$c \rightarrow \lambda$$

    Para $c \subseteq O^+$ representando el multiconjunto de spikes que consumirá la regla al activarse sin producir activación (no enviará spikes por sus sinapsis). Es importante notar que aunque no tiene expresión regular, no está limitada como las reglas de impulso, es decir, puede ejecutarse siempre que haya al

menos c spikes y mantenemos el orden de prioridad de los SNP Systems, siendo las reglas de olvido siempre menos prioritarias que las de impulso.

d) $syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m, out\} \times L$ conjunto de sinapsis (conexiones entre neuronas), $(i, i, l) \notin syn$ para $\forall 1 \leq i \leq m$ y para $\forall l \in L$ pues no existen bucles y $out \notin O \wedge out \notin L$ indicando que su contenido se envía al entorno.

e) $in \in \{1, 2, ..., m\}$ neurona de entrada (se puede omitir en sistemas generadores).

A lo largo de este trabajo asumiremos siempre que el modelo se ejecuta con maximo paralelismo, es decir, en cada instante de tiempo se activará toda neurona que tenga alguna regla aplicable (el contenido de la neurona cumple su expresión regular y hay al menos tantos spikes como los que consume la regla). Si hay varias reglas aplicables las de impulso tendrán prioridad sobre las de olvido, si hay varias reglas en conflicto con la misma prioridad se elegirá una de manera indeterminista, se ejecutará la regla elegida tanto como sea posible y una vez la dicha regla no pueda seguir ejecutándose se comprobará si alguna de las restantes puede y se repetirá el proceso.

Si se activa una regla con retardo, la neurona quedará bloqueada al igual que pasa con los SNP Systems, si en un mismo instante se lanzan varias reglas de bloqueo, la neurona quedará bloqueada en función del máximo valor de retardo de las reglas ejecutadas, pero las que cuenten con un valor inferior emitirán sus spikes en función de su propio retardo.

Todo esto se realiza de manera paralela en todas las neuronas y supondremos que todo impulso enviado en un instante de tiempo no llega a su destino hasta el inicio del instante siguiente.

Por último cabe destacar que consideraremos que la computación ha acabado cuando en un instante de tiempo no haya ninguna neurona en retardo y no se haya aplicado ninguna regla.

Respecto a la salida del sistema podemos leerla de cuatro formas distintas:

a) **parada**: El resultado de la computación son los spikes que han sido enviados al entorno a lo largo de dicha computación, por lo que leeremos el resultado cuando el sistema alcance un estado de parada.

b) **parada multi-canal**: Al igual que en el modo halt interpretaremos que la salida del modelo son los spikes enviados al entorno durante la ejecución, pero los agruparemos en función de los canales por los que han sido enviados.

c) **temporal**: Este modo es el empleado para los sistemas generadores, pues no es necesario alcanzar un estado de parada, la salida del sistema se leerá como la secuencia de spikes enviados al entorno en cada instante de tiempo.

d) **temporal multi-canal**: Empleado para el procesamiento de señales, pues nos permite leer la salida como la secuencia de spikes enviados al entorno en cada instante de tiempo por cada uno de los canales.

Para terminar de ver el funcionamiento del modelo, analizaremos el ejemplo que se muestra en la figura 4.

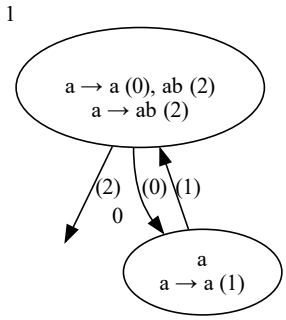

Figure 4: Ejemplo de SNP-MC System con múltiples spikes

En el primer instante de tiempo el modelo solo puede ejecutar la regla de la neurona 0, enviando un spike a a la neurona 1.

En el segundo instante se elegirá de manera indeterminista entre las reglas:

a) $a \rightarrow a(0), ab(1)$ en cuyo caso se enviarán los spikes ab al entorno y un spike a a la neurona 1, por lo que en el siguiente instante repetiríamos el primer paso.

b) $a \rightarrow ab(1)$ en cuyo caso se enviarán los spikes ab al entorno y la ejecución terminará.

Como podemos observar el sistema es equivalente al empleado en el ejemplo de los sistemas P en la sección 2.4, un generador de los multiconjuntos definidos por el lenguaje $\{a^n b^n : n > 0\}$.

4 Simulador

La primera herramienta que creamos para facilitar el trabajo con el modelo fue un simulador, este simulador está implementado en *python* y permite construir un *sistema P con impulsos neuronales multi-canal y múltiples spikes*, configurarlos y ejecutarlos.

Para enseñar el funcionamiento de este simulador, podemos acudir a la figura 4, donde construimos una operación add similar a la de una máquina de registros.

```python
# Construimos el modelo
snp = SNPSystem[int, int]()

# Definimos la neurona de entrada
snp.set_input(0)

# Inicializamos el contenido de las neuronas
snp.add_symbols(1, *['1']*3)
snp.add_symbols(2, *['1']*2)

# Inicializamos las sinapsis y definimos a que canal pertenecen
snp.add_channel(1, 5, 1)
snp.add_channel(2, 0, 2)
snp.add_channel(2, 5, 2)
snp.add_channel(3, 5, 3)
snp.add_channel(4, 5, 4)
snp.add_channel(5, 2, 5)

# Añadimos las reglas de activación
snp.add_rule(0, 'a', Multiset(['a']), {2: Multiset(['a'])})
snp.add_rule(2, '1*a', Multiset(['1']), {5: Multiset(['1'])})
snp.add_rule(2, '1*a', Multiset(['a']), {5: Multiset(['a'])})
snp.add_rule(5, '1*a', Multiset(['1']), {2: Multiset(['1']), 1: Multiset(['1'])})
snp.add_rule(5, '1*a', Multiset(['a']), {3: Multiset(['a'])})
snp.add_rule(5, '1*a', Multiset(['a']), {4: Multiset(['a'])})

# Ejecutamos el modelo
result = snp.run(['a'], render_steps=True)
```

Figure 5: Ejemplo de código para crear un sistema

Como se puede observar el uso del simulador es bastante intuitivo y su uso se divide en 6 pasos:

a) Construcción de un modelo.

b) Especificación de las neuronas de entrada, al ejecutar el modelo se podrá facilitar un multiconjunto para añadir a la membrana de entrada.

c) Inicializamos el contenido de las neuronas, a la función *add_symbols*, se le debe pasar cada símbolo de manera individual como parámetro (por ello se desestructura la lista).

d) Añadimos las sinapsis a sus respectivos canales, siendo el primer parámetro el canal, el segundo la neurona de salida y el tercero la neurona destino.

e) Para acabar con la definición del modelo añadimos las reglas de activación, con los parámetros neurona, expresión regular, multiconjunto consumido, multiconjuntos enviados (en forma de diccionario {*canal* : *multiconjunto*}) y opcionalmente delay como

último parámetro. Si el diccionario de multiconjuntos enviados está vacío se entenderá que es una regla de olvido y se ignorará la expresión regular.

f) Una vez el modelo está definido se puede ejecutar empleando el método *run*, cuyo primer parámetro es el multiconjunto que se enviará a la membrana de entrada y el parámetro opcional *render_steps* permite renderizar cada uno de los pasos de ejecución, como se puede ver en la figura 6.

En este ejemplo no se ha enviado nada al entorno, si fuera necesario, se puede especificar una "neurona" de salida con *snp.set_output(n)*, donde la neurona *n* pasará a representar el entorno.

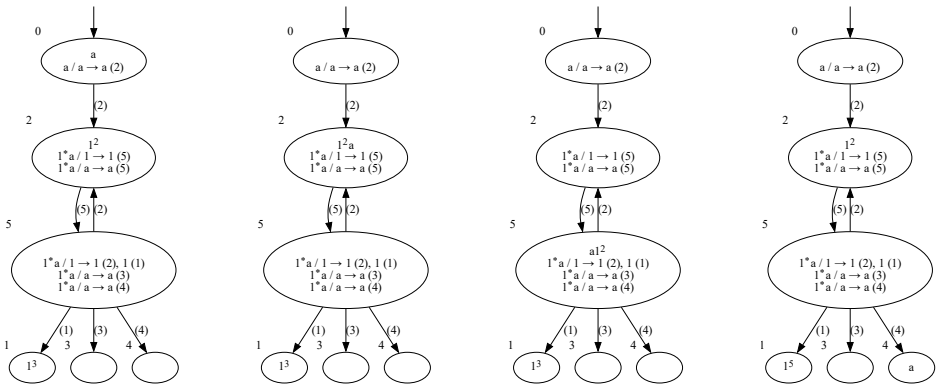

Figure 6: Renderizado de los pasos de ejecución

5 Lenguaje de especificación

Aunque gracias al simulador ya podemos definir modelos y probarlos, la forma de definirlos es un poco engorrosa y puede llegar a dificultar la lectura, por ello diseñamos un lenguaje de especificación basado en Python, que permite definir estos modelos de una manera más cómoda y natural sin tener que interactuar directamente con el simulador.

Un ejemplo de nuestro lenguaje de especificación para definir un modelo como el visto en la sección anterior, sería el de la figura siguiente.

```
# Definimos la neurona de entrada
input([0])

# Inicializamos el contenido de las neuronas
[1] = {'1'} * 3
[2] = {'1'} * 2

# Inicializamos las sinapsis y definimos a que canal pertenecen
<1> [5] --> [1]
<2> [0] --> [2]
<2> [5] --> [2]
<3> [5] --> [3]
<4> [5] --> [4]
<5> [2] --> [5]

# Añadimos las reglas de activación
[0] 'a' / {'a'} --> {'a'} <2>

[2] '1'* 'a' / {'1'} --> {'1'} <5>
[2] '1'* 'a' / {'a'} --> {'a'} <5>

[5] '1'* 'a' / {'1'} --> {'1'} <2>, {'1'} <1>
[5] '1'* 'a' / {'a'} --> {'a'} <3>
[5] '1'* 'a' / {'a'} --> {'a'} <4>
```

Figure 7: Ejemplo del lenguaje de especificación

Como se puede ver en el ejemplo, cada línea tiene a su equivalente en el código del simulador, pero el lenguaje de especificación resulta mucho más legible y cómodo de utilizar.

A continuación describiremos las distintas operaciones que permite este pseudo-lenguaje.

5.1 Tipos de datos

Los dos tipos de datos básicos son los enteros, los símbolos (representados como cadenas Python) y los multiconjuntos (representados como sets Python). Los tipos de datos básicos se pueden almacenar en variables, por ejemplo `n = 12`.

Además podemos trabajar con membranas (neuronas) que se representan como un entero entre corchetes (Ej.: `[1]`), aunque en lugar de un literal se puede emplear una variable que contenga a ese literal, por ejemplo partiendo de la variable `n` definida anteriormente, podemos acceder a la membrana 12 mediante `[n]`.

5.2 Operador de asignación

Los valores siempre se pasan por copia, por lo que si asignas un valor a una variable y modificas el valor de dicha variable, no modificas el valor original.

Las membranas son un caso especial, pues no se trata de un tipo básico. Si tratamos de asignar un valor a una membrana, lo que haremos en realidad es establecer el multiconjunto inicial de dicha membrana (sobra decir que solo se le puede asignar un valor de tipo multiconjunto). Si por el contrario tratamos de almacenar una membrana en una variable, copiaremos su multiconjunto.

Por lo tanto

```
[1] = {'a'}
b = [1]
```

es equivalente a

```
[1] = {'a'}
b = {'a'}
```

5.3 Operadores aritméticos

En los siguientes apartados iremos explicando los distintos operadores que permite cada tipo de datos, a nivel general solo cabría destacar que el orden de las operaciones es el mismo que en la mayoría de los lenguajes:

$$Unarias(-) > Multiplicativas(/ * \%) > Aditivas(+-) > Lógicas(|\&)$$

5.3.1 Enteros

Los enteros constan de los operadores básicos suma(+), resta(-), multiplicación(*), división(/) y módulo(%). El lenguaje no contempla números decimales, así que la división será entera (equivalente al operador // en python).

5.3.2 Símbolos (strings)

Los símbolos solo permiten dos operadores, la concatenación (+) que permite concatenar tanto símbolos como un símbolo y un entero.

```
'a' + 'a'    # 'aa'
'a' + 1      # 'a1'
1   + 'a'    # '1a'
```

Y el producto (*), que solo se contempla entre un símbolo como parte izquierda y un entero como parte derecha.

```
'ab' * 3      # 'ababab'
2    * 'a'    # Error
```

5.4 Multiconjuntos

Sobre multiconjuntos está definida la unión(|), concatenación (+), intersección (&), diferencia (-) y producto (*). Las primeras cuatro operaciones solo se pueden aplicar sobre dos multiconjuntos y se definen como:

- La unión (|) se define formalmente como

$$a \cup b = (A_a, m_a) \cup (A_b, m_b) = (A_a \cup A_b, m_{a \cup b})$$

donde

$$m_{a \cup b}(s) = max(m_a(s), m_b(s))$$

- La concatenación (+) se define formalmente como

$$a + b = (A_a, m_a) + (A_b, m_b) = (A_a \cup A_b, m_{a+b})$$

donde

$$m_{a+b}(s) = m_a(s) + m_b(s)$$

- La intersección (&) se define formalmente como

$$a \cap b = (A_a, m_a) \cap (A_b, m_b) = (A_a \cap A_b, m_{a \cap b})$$

donde

$$m_{a \cap b}(s) = min(m_a(s), m_b(s))$$

- La diferencia (-) se define formalmente como

$$a - b = (A_a, m_a) - (A_b, m_b) = (A_a - A_b, m_{a-b})$$

donde

$$m_{a-b}(s) = \begin{cases} m_a(s) - m_b(s) & \text{si } m_a(s) \geq m_b(s) \\ 0 & \text{si no} \end{cases}$$

Y por último tendriamos el producto, que debe tener un multiconjunto como parte izquierda y un entero como parte derecha, y podríamos definirlo formalmente como

$$a \times n = (A, m) \times n = (A, m_{a \times n})$$

donde

$$m_{a \times n}(s) = m(s) \times n$$

Todas estas operaciones se verían de la siguiente manera en el código:

```
{'a', 'b'} | {'b', 'b'}          # {'a', 'b', 'b'}
{'a', 'b'} + {'b', 'b'}          # {'a', 'b', 'b', 'b'}
{'a', 'b'} & {'a', 'a'}          # {'a'}
{'a', 'a', 'b'} - {'a', 'c'}     # {'a', 'b'}
{'a', 'b'} * 2                   # {'a', 'a', 'b', 'b'}
```

5.5 Neurona de entrada y salida al entorno

Si queremos marcar una neurona como entrada, podemos emplear la función `input()` , que recibe como único parámetro la neurona de entrada (Ej.: `input([2])`).

Por otro lado, para referirnos al entorno o neurona de salida, podemos usar la palabra reservada `out` .

5.6 Canales

Para definir un canal y/o añadirle una sinapsis, empleamos la sintaxis:

`canal salida --> destino`

Es importante tener en cuenta que el canal debe ir entre " <" y " >", y que `salida` y `destino` deben ser membranas. Por ejemplo, si queremos crear dos sinapsis en el canal 0, que salen de la membrana 1, una llega a la membrana 2 y la otra al entorno, escribiríamos:

```
<0> [1] --> [2]
<0> [1] --> out
```

5.7 Reglas de activación

Para añadir una regla de activación a una membrana empleamos la sintaxis:

`membrana regex / consumido -->` $enviado_1$ $canal_1$`,` $enviado_2$ $canal_2$`, ... : delay`

Donde `membrana` es la membrana donde se encuentra la regla, `regex` es la expresión regular de dicha regla, `consumido` es el multiconjunto que consume, $enviado_i$ es el multiconjunto que envía por el canal $canal_i$ y `delay` es el delay de la regla.

La regla de activación debe tener como mínimo un par $enviado_1$ $canal_1$, pero hay otra información que se puede omitir, como la expresión regular (en cuyo caso será igual al multiset consumido) y el delay ($delay = 0$), dejando la regla con la forma:

`membrana consumido -->` $enviado_1$ $canal_1$`,` $enviado_2$ $canal_2$`, ...`

Es importante notar que al omitir la expresión regular también se ha omitido la contrabarra, si se deja, el sistema interpretará que la expresión regular es λ, lo que hará la regla inaplicable.

5.7.1 Expresión regular

La expresión regular permite el uso de paréntesis, el operador uno o más (+) y el operador cero o más (*), para especificar los símbolos, se deben poner entre comillas.

Ejemplos:

```
'a'+ 'b'          # a+b
('b''a'+)'b'*     # ba+b*
('b''a')+'b'*     # (ba)+b*
```

Notese que una expresión regular es en ocasiones indistinguible de una operación entre símbolos (como en el primer caso), por lo tanto se asumirá que todo lo que se encuentra en una regla entre la membrana y la contrabarra es una expresión regular.

5.8 Reglas de olvido

Para las reglas de olvido se hace uso de la palabra reservada `lambda`, siguiendo la sintaxis:

```
membrana consumido --> lambda
```

Donde `membrana` es la membrana donde se encuentra la regla y `consumido` es el multiconjunto que consume.

6 Interprete

Una vez hemos creado un lenguaje de especificación y un simulador, queda el paso evidente, interpretar ese lenguaje para pasárselo al simulador, y eso es lo que hemos hecho. Hemos creado un Tree-Walk interpreter siguiendo los primeros 8 capítulos de [8] de Robert Nystrom, para poder interpretar el código de especificación y ejecutar el modelo con el simulador, todo combinado en una única herramienta.

A través de las siguientes sub-secciones analizaremos los diferentes módulos de los que se compone el interprete, así como su integración con el simulador en una única herramienta.

6.1 Scanner

El escaneado (o tokenizado) es el primer paso para interpretar un lenguaje, consiste en transformar el texto en una lista de tokens, para que el parser pueda interpretar.

Por ejemplo, a partir del código siguiente:

```
# Definimos la neurona de entrada
input([0])

# Inicializamos el contenido de las neuronas
[1] = {'1'} * 3
```

El tokenizador devolvería algo como esto:

```
ID(input) OPEN_PARENTHESIS OPEN_MEMBRANE NUMBER(0) CLOSE_MEMBRANE
    CLOSE_PARENTHESIS EOL
OPEN_MEMBRANE NUMBER(1) CLOSE_MEMBRANE EQUAL OPEN_SET SYMBOL('1') CLOSE_SET MULT
    NUMBER(3) EOL
EOF
```

La implementación del tokenizador es sencilla, se itera carácter a carácter y se van generando los tokens, esto solo varía cuando hay tokens con un valor variable (como un número o el identificador de una variable), en ese caso se avanzan varios símbolos de golpe hasta que encontramos el final de esa expresión y la guardamos en el token, para poder recuperarla posteriormente.

6.2 Parser

Una vez tenemos el texto tokenizado, es hora de estructurarlo, para ello implementamos un parser recursivo descendente que generará un árbol de sintaxis abstracta (AST) para que el interprete pueda recorrer e interpretar.

Durante el desarrollo de este modulo encontramos más dificultades, en primer lugar es importante el orden en el que se parsean las expresiones, pues eso determinará el orden de ejecución y prioridad de los operandos.

En segundo lugar, habrá ocasiones en las que una expresión presente ambigüedad y el parser no sepa como interpretarla, para ello se ha creado un sistema de checkpoints, esto permite que cuando encontremos una ambigüedad elijamos una de las posibles interpretaciones y si resulta que no es la adecuada, volvemos al punto donde encontramos dicha ambigüedad y probamos otro camino, un ejemplo de esto sería la expresión siguiente:

```
[1 + 2] = {'a'}
[1 + 2] + {'a'}
```

Que una vez tokenizado quedaría de la siguiente manera:

```
OPEN_MEMBRANE NUMBER(1) ADD NUMBER(2) CLOSE_MEMBRANE EQUAL OPEN_SET SYMBOL('a')
    CLOSE_SET EOL
OPEN_MEMBRANE NUMBER(1) ADD NUMBER(2) CLOSE_MEMBRANE ADD OPEN_SET SYMBOL('a')
    CLOSE_SET EOL
EOF
```

El inicio de ambas expresiones es idéntico (`[1 + 2]`) , pero la primera es una asignación y la segunda una suma, cada una de estas operaciones tiene una prioridad diferente (la asignación es mucho más prioritaria), por lo que deben parsearse en momentos distintos, en la primera línea no habría problemas pues lo primero que se parsea en una expresión es la igualdad.

Pero en la segunda línea el sistema parsearía la parte izquierda esperando encontrarse una igualdad y al llegar al operador se encontraría con una suma, pero ya no puede volver atrás, porque la longitud en tokens de una membrana es variable pues puede contener expresiones. Para esto son los checkpoints, al iniciar el parseo de la asignación se guarda la posición en la lista de tokens del inicio de la expresión, de manera que si falla a mitad, puede recuperar la posición inicial y pasarle el control a la función que parsea el siguiente operador (por orden de prioridad).

El AST generado para el ejemplo anterior sería:

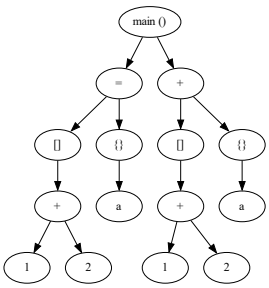

Figure 8: Árbol de sintaxis abstracta

Aunque ambas ramas son iguales, esto se debe a que es un ejemplo sencillo, pero para expresiones más complejas el orden del parsing si que implicaría una diferencia relevante.

6.3 Intérprete

El intérprete recorre el árbol de sintaxis abstracta generado por el parser, calcula los valores de las expresiones y ejecuta las instrucciones.

Al iniciar el intérprete genera un modelo con el simulador visto en la sección 4 y cuando ejecuta una instrucción lo que hace es llamar a la función equivalente en el simulador. El intérprete también se encarga de hacer la comprobación de tipos en tiempo de ejecución, abortando y devolviendo un error en caso de encontrar una operación no definida.

Para gestionar la memoria (variables y contenido de membranas), se diseñó un submódulo llamado *MemoryManager*, este modulo nos permite trabajar con instancias de una clase llamada *Data*, que gestionarán internamente los valores y tipos. De este modo, en lugar de trabajar con los tipos de Python, trabajaremos con este "wrapper" que nos facilitará el trabajo y nos permitirá interactuar con el *MemoryManager*.

6.4 Herramienta final

Para acabar, combinamos todos estos módulos en una aplicación por linea de comandos disponible en Github en el repositorio https://github.com/RodrigoLlanes/SNP-MC-MS-Systems-simulator con los siguientes parámetros:

```
Usage: main.py [OPTIONS] SRC

Options:
  -i, --input TEXT
  --separator TEXT
  --no-strip
  --render
  --render-path TEXT
  -r, --repeat INTEGER
  -m, --mode [halt|halt-mc|time|time-mc]
  --max-steps INTEGER
```

Donde aparte de *SRC*, que es la ruta hasta el fichero donde se encuentra la especificación del modelo que se desea interpretar, se pueden especificar los siguientes parámetros opcionales:

- **input**: Una cadena de caracteres que indica los símbolos que se encuentran en la neurona de entrada al inicio de la computación.
- **separator**: El carácter de separación empleado en *input*, por defecto la coma.
- **no-strip**: Por defecto se eliminarán los espacios en blanco de los símbolos de entrada, si se usa esta flag, se mantendrán.
- **render**: Al emplear esta flag se le indica a la aplicación que se desean renderizar cada una de las iteraciones del sistema.
- **render-path**: Directorio donde se desean guardar los renders (solo se emplea si se ha usado la flag *render*).
- **repeat**: Número de veces que se desea repetir la ejecución.
- **mode**: Modo de lectura de la salida (*halt* por defecto).
- **max-steps**: Máximo número de iteraciones (fuerza la parada de ejecuciones que la superen).

La figura 6 de la sección 4 es un ejemplo del output de la aplicación con el comando:

```
python main.py -i a -r tests/src/reg-add.txt
```

Siendo *tests/src/reg-add.txt* la ruta al fichero donde se encuentra el código mostrado en la misma sección.

7 Resultados teóricos

A lo largo de las siguientes subsecciones presentaremos los resultados teóricos obtenidos con el modelo, empezando por la universalidad, pasando por la capacidad de simular otros modelos y acabando con la unificación de estos.

7.1 Universalidad

Como ya demostró Marvin Lee Minshy en [7], se puede obtener la universalidad con una máquinas de registros con únicamente dos registros, por lo tanto para demostrar la universalidad de nuestro modelo seguiremos los pasos de Mihai Ionescu, Gheorghe Păun y Takashi Yokomori en [4] simulando una máquina de registros.

Para ello, haremos uso principalmente de un alfabeto de cuatro spikes $O = \{a, h, c, d\}$, en cada paso de ejecución habrá como máximo un spike a en todo el sistema, este hará la función de "puntero" indicando la instrucción activa en ese momento, lo que nos permitirá controlar el flujo de ejecución y asegurará la secuencialidad. Por el contrario el spike h disparará la regla de parada.

Por otro lado el símbolo c hará la función de contador, por lo tanto si en una neurona tenemos c^4 equivale a tener el valor 4 en el registro que esa neurona representa. Y por último el símbolo d activará la regla de decremento de la neurona en la que se encuentre, representando una operación SUB.

7.1.1 Instrucción ADD

Podemos observar una implementación de la instrucción ADD en la figura 9.

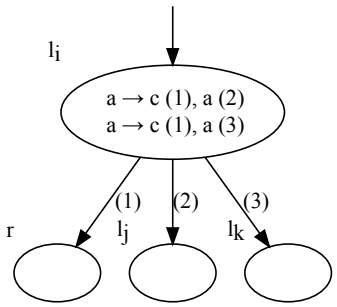

Figure 9: Implementación de la instrucción ADD

El funcionamiento es sencillo, al recibir un spike a en la neurona l_i (que representa a la instrucción con la misma etiqueta), se ejecuta una de las dos reglas de manera indeterminista, ambas enviarán un spike c a la neurona r (que representa el incremento de dicho registro) y un spike a a l_j o l_k (lo que implica un salto a dicha instrucción) en función de la regla escogida .

Si queremos simular una máquina determinista lo único que tenemos que hacer es eliminar una de las dos reglas de activación.

7.1.2 Instrucción SUB

A continuación presentamos la representación de la instrucción *SUB* en la figura 10.

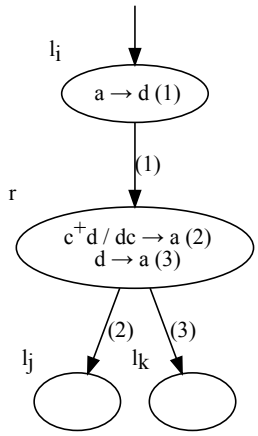

Figure 10: Implementación de la instrucción SUB

Al recibir un spike a la neurona l_i lo consume y envía un spike de decremento d al registro r, si en el registro existe algún spike c se consume uno de esos spikes y el de decremento, envíando la señal de activación a a la instrucción l_j, en caso contrario se consume solo el spike d y se activa la instrucción l_k.

Nótese que la instrucción *SUB* está integrada dentro de la neurona que representa el registro, por lo que si se desea que varias instrucciones *SUB* decrementen el mismo registro, se deben incluir variaciones en los símbolos y etiquetas de los canales. Por ejemplo, si quisiéramos añadir una nueva instrucción $l_{i'} = (SUB(r), l_{j'}, l_{k'})$, obtendríamos algo similar a 11.

Y este proceso se puede repetir tantas veces como sea necesario para generar múltiples instrucciones *SUB* aplicadas a un mismo registro.

7.1.3 Instrucción HALT

Por último implementamos la instrucción *HALT*, a la que añadiremos la capacidad de enviar el contenido de un registro al entorno (el output). Una posible implementación de esta instrucción está representada en la figura 12.

Al recibir un spike a la neurona l_h envía un spike h al "registro" cuyo contenido queremos devolver como output, al recibir dicho spike consumirá todos sus spikes contador (c) y enviará una copia el entorno, cuando ya solo quede el spike h aplicará la regla de olvido y la ejecución parará (pues no se ha generado ningún spike a que pueda activar otra instrucción).

Tras esto podemos simular cualquier máquina de registros y por lo tanto nuestro modelo puede computar los lenguajes Recursivamente Enumerables, es decir cualquier problema que pueda resolver una máquina de Turing.

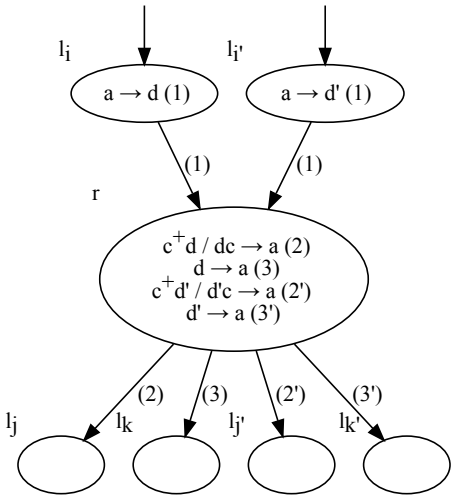

Figure 11: Ejemplo de múltiples instrucciones SUB sobre el mismo registro

7.2 Simulación de SNP-Systems

El primer modelo que vamos a simular es el SNP-System [4], su simulación es trivial pues nuestro modelo deriva de él e incorpora todas sus propiedades, de todos modos es interesante ver su simulación pues más adelante la emplearemos para unificar los SNP-Systems con otros modelos.

Para la simulación emplearemos un único canal al que llamaremos c_s y un único spike al que llamaremos a_s, de este modo las reglas de un SNP-System, que son de la forma:

$$E/a^c \rightarrow a; d$$

se pueden expresar como:

$$E/a_s^c \rightarrow a_s(c_s); d$$

Además de esta modificación, se copiarán las neuronas del modelo original, cualquier sinapsis de dicho sistema se incluirá en el canal c_s y el contenido original de cada neurona se incluirá en su respectiva copia sustituyendo los spikes a por a_s.

7.3 Simulación de máquinas de virus

7.3.1 Descripción formal

Las máquinas de virus (VM para abreviar) son un modelo computacional basado en los virus y sus mecanismos de propagación, fue presentado por Xu Chen, Mario J Pérez Jiménez, Luis Valencia Cabrera and Beizhan Wang y Xiangxiang Zeng en [2] donde se define un VM de grado $(p, q), p \geq 1, q \geq 1$ como una tupla de la forma:

$$\Pi = (\Gamma, H, I, D_H, D_I, D_C, n_1, ..., n_p, i_1, h_{out})$$

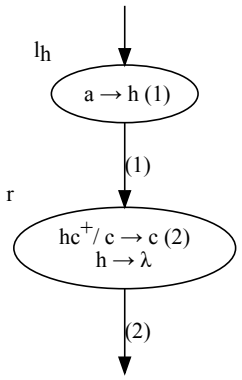

Figure 12: Implementación de la instrucción HALT

donde:

a) $\Gamma = \{v\}$ es un alfabeto con un único símbolo al que llamaremos virus.

b) $H = \{h_1, ..., h_p\}$ es un conjuntos ordenado de anfitriones o huéspedes tal que $v \notin H$.

c) $I = i_1, ..., i_q$ es un conjunto ordenado de unidades de instrucciones de control tal que $v \notin I$ y $H \cap I = \emptyset$.

d) $D_H = (H \cup \{h_{out}\}, E_H, w_H)$ es un grafo dirigido ponderado, donde $E_H \subseteq H \times (H \cup \{h_{out}\})$, $(h, h) \notin E_H \forall h \in H$, el grado de salida de h_{out} es igual a cero y w_H es una función de la forma $w_H : E_H \to \mathbb{N}^+$. Este grafo representa los canales de transmisión entre los distintos huéspedes, todos los canales están cerrados por defecto hasta que son abiertos por una instrucción de control. Por otro lado la función de pesos w_H indica el número de virus que serán transmitidos/replicados al anfitrión objetivo del canal.

e) $D_I = (I, E_1, w_1)$ es un grafo dirigido ponderado, donde $E_I \subseteq I \times I$, w_I es una función de la forma $w_I : E_I \to \mathbb{N}^+$ y el grado de salida de todo vértice $i_j \in I$ es menor o igual a 2. Los ejes de dicho grafo representan los caminos de transferencia de instrucciones.

f) $G_C = (V_C, E_C)$ es un grafo no dirigido bipartito, donde $V_C = I \cup E_H$, siendo $\{I, E_H\}$ la partición asociada con él (es decir, cada eje conecta un elemento de cada conjunto). Además, el grado de cada vértice $i_j \in I$ es menor o igual a 1. Representando este grafo la red de instrucción-canal, indicando una relación de control entre una instrucción i_j y un canal de transmisión $(h_s, h_{s'})$: cuando la instrucción i_j se activa el canal $(h_s, h_{s'})$ se abre permitiendo la transmisión de un virus del anfitrión h_s al anfitrión $h_{s'}$, replicándose $w_{s,s'}$ veces en el destino.

g) $n_j \in \mathbb{N} (1 \leq j \leq p)$ indicando el número de virus que contiene originalmente el anfitrión h_j.

h) $h_{out} \notin I \cup \{v\}$ denotando la región de salida, se denota como h_0 en caso de que $h_{out} \notin H$ en cuyo caso nos referiremos a el como el entorno.

Para poder visualizar correctamente la estructura de una máquina de virus, emplearemos la figura 13, una variación del ejemplo que usan en [2] de una máquina de virus de grado $(4, 6)$ es decir 4 huéspedes y 6 unidades de control.

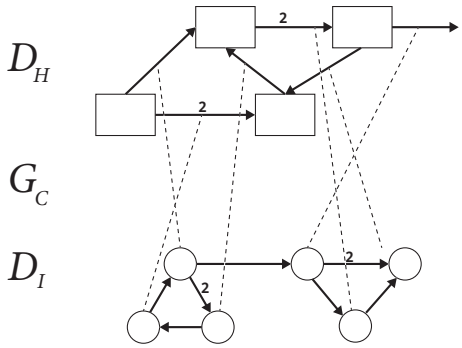

Figure 13: Estructura de una máquina de virus

Además de su estructura es importante entender su funcionamiento, para ello debemos tener en cuenta que en un instante t el estado de la máquina puede describirse como $C_t = (a_{1,t}, ..., a_{p,t}, u_t, e_t)$ siendo $a_{i,t} (1 \leq i \leq p)$ números naturales que indican el número de virus en el anfitrión i, u_t la unidad de instrucción de control activada (cuyo efecto quedará reflejado en el estado C_{t+1}) y e_t un número natural que representa el número de virus en el entorno o región de salida.

Cabe destacar que en el estado inicial C_0 la instrucción activada será i_1 y que existe una "instrucción" especial $\# \notin H \cup h_0 \cup I$ que indica parada.

Dado un estado $C_t = (a_{1,t}, ..., a_{p,t}, u_t, e_t)$ el estado C_{t+1} se obtiene siguiendo siguientes pasos:

a) Asumiendo que C_t no es una configuración de parada tenemos $u_t \in I$, por lo que la unidad de instrucción de control u_t se activará.

b) Si la instrucción u_t está conectada al canal $(h_s, h_{s'})$ dicho canal se abrirá y:

   - Si $a_{s,t} \geq 1$ un único virus será consumido del anfitrión h_s y $w_{s,s'}$ copias de v se producirán en el anfitrión $h_{s'}$ actualizando los valores del estado $a_{s,t+1} = a_{s,t} - 1$ y $a_{s',t+1} = a_{s',t} + w_{s,s'}$. En cambio si $s' = out$, el estado se actualizará $a_{s,t+1} = a_{s,t} - 1$ y $e_{t+1} = e_t + w_{s,s'}$.
   - Si $a_{s,t} = 0$ no hay transmisión, por lo que no se producirá ningún virus en $h_{s'}$.

c) Si la instrucción u_t no está conectada a ningún canal, no habrá transmisión de virus.

d) Por último la unidad de control de instrucción seleccionada para el siguiente estado $u_{t+1} \in I \cup \{\#\}$ se obtendrá de la siguiente manera:

- Si el grado de salida de u_t es igual a 2, existen dos instrucciones diferentes $u_{t'}$ y $u_{t''}$ tales que $(u_t, u_{t'}) \in E_I$ y $(u_t, u_{t''}) \in E_I$.
  - Si la instrucción u_t está conectada a un canal $(h_s, h_{s'})$:
    - Si $a_{s,t} \geq 1$ (hay virus en h_t en el estado C_t) u_{t+1} es la instrucción correspondiente al camino de mayor peso $(max(w_{t,t'}, w_{t,t''}))$. Si $w_{t,t'} = w_{t,t''}$ la siguiente instrucción se elige de manera indeterminista.
    - Si $a_{s,t} = 0$ (no hay virus en h_t en el estado C_t) u_{t+1} es la instrucción correspondiente al camino de menor peso $(min(w_{t,t'}, w_{t,t''}))$. Si $w_{t,t'} = w_{t,t''}$ la siguiente instrucción se elige de manera indeterminista.
  - Si la instrucción u_t no está conectada a ningún canal la siguiente instrucción se elige de manera indeterminista.$(h_s, h_{s'})$:
- Si el grado de salida de u_t es igual a 1, el sistema se comporta de manera determinista y u_{t+1} será la unidad de control de instrucción que cumpla $(u_t, u_{t+1}) \in E_I$.
- Si el grado de salida de u_t es igual a 0, $u_{t+1} = \#$ y la configuración C_{t+1} será una configuración de parada.

7.3.2 Simulación

Una vez comprendemos el funcionamiento de las máquinas de virus podemos plantear una forma de simularlas, para ello construiremos un SNP-MC-System con múltiples spikes que simule una máquina de virus mediante el siguiente algoritmo:

a) Creamos un spike v que representa a un virus y tres spikes auxiliares s, f y a_i.

b) Creamos una neurona h_i para todo $h_i \in H$ y una neurona i_j para todo $i_j \in I$.

c) Para cada arista (h_s, h'_s) de D_H, asociada con una unidad de control de instrucción i_k:

   (a) Creamos un canal c_{h_s,h'_s} que representa el canal de comunicación entre huéspedes.

   (b) Creamos otro canal c_{i_k} que representa la relación de control entre el canal (h_s, h'_s) y la instrucción i_k.

   (c) Añadimos al canal c_{h_s,h'_s} una sinapsis que va de la neurona h_s a la neurona h'_s.

   (d) Añadimos al canal c_{i_k} una sinapsis que va de la neurona h_s a la neurona i_k y otra en sentido contrario.

   (e) Creamos un spike a_{h_s,h'_s} que representa la señal de activación del canal (h_s, h'_s).

   (f) Añadimos a la neurona h_s dos reglas de activación de la forma:

$$a_{h_s,h'_s}v^+/va_{h_s,h'_s} \rightarrow v^{w_{h_s,h'_s}}(c_{h_s,h'_s}), s(c_{i_k})$$

   Indicando que si se recibe una señal de activación y hay virus en el huésped, consume un virus y dicha señal de activación, enviando w_{h_s,h'_s} virus a s' y un spike s (succes) a la instrucción que la ha activado.

$$a_{h_s,h'_s}/a_{h_s,h'_s} \rightarrow f(c_{i_k})$$

Indicando que si se recibe una señal de activación y no hay virus en el huésped, consume la señal de activación y envía un spike f (failure) a la instrucción que la ha activado.

d) Para cada unidad de control de instrucción i_k:

- Si el grado de salida de i_k es igual a 2, existen dos instrucciones diferentes $i_{k'}$ y $u_{k''}$ tales que $(i_k, i_{k'}) \in E_I$ y $(i_k, i_{k''}) \in E_I$.
  - Si la instrucción i_k está conectada a un canal $(h_s, h_{s'})$:
    * Creamos una regla $a_i/a_i \to a_{h_s,h_{s'}}(c_{i_k})$, que al recibir un spike de activación de instrucción a_i lo consume y envía un spike de activación a ese canal.
    * Creamos dos canales $c_{i_k,i_{k'}}$ y $c_{i_k,i_{k''}}$.
    * Si $w_I(i_{k'}) = w_I(i_{k''})$ creamos cuatro reglas de la forma:

$$s/s \to a_i(c_{i_k,i_{k'}})$$

$$f/f \to a_i(c_{i_k,i_{k'}})$$

$$s/s \to a_i(c_{i_k,i_{k''}})$$

$$f/f \to a_i(c_{i_k,i_{k''}})$$

      Para que se elija la siguiente instrucción de manera indeterminista.
      En caso contrario, asumiendo que $i_{max} = \text{argmax}_{i\in\{i_{k'},i_{k''}\}} w_i$ y $i_{min} = \text{argmin}_{i\in\{i_{k'},i_{k''}\}} w_i$, crearemos dos reglas:

$$s/s \to a_i(c_{i_k,i_{max}})$$

$$f/f \to a_i(c_{i_k,i_{min}})$$

      De modo que si en el huésped había virus se saltará a la instrucción con camino de mayor peso y en caso contrario a la de menor.
  - Si la instrucción i_k no está conectada a ningún canal:
    * Creamos dos canales $c_{i_k,i_{k'}}$ y $c_{i_k,i_{k''}}$.
    * Crearemos dos reglas:

$$a_i/a_i \to a_i(c_{i_k,i_{k'}})$$

$$a_i/a_i \to a_i(c_{i_k,i_{k''}})$$

      De modo que la siguiente instrucción se elija de manera indeterminista.
- Si el grado de salida de i_k es igual a 1 y $i_{k'}$ es la unidad de control de instrucción que cumple $(i_k, i_{k'}) \in E_I$, se creará una regla de la forma $a_i/a_i \to a_i(c_{i_k,i_{k'}})$.
- Si el grado de salida de i_k es igual a 0, se creará una única regla de olvido de la forma $a_i/a_i \to \lambda$.

e) Para cada $n_j (1 \le j \le p)$ inicializaremos el contenido de la neurona h_j a v^{n_j}.

f) Añadimos el spike a_i a la neurona i_1.

g) Por último es importante tener en cuenta que la neurona h_{out} será reemplazada de todas las sinapsis por el símbolo *out* que representa al entorno.

Siguiendo estos pasos somos capaces de simular cualquier máquina de virus empleando SNP-MC-Systems con múltiples spikes.

7.4 Simulación de Sistemas P basados en impulsos neuronales con plásmidos

7.4.1 Descripción formal

Los sistemas P basados en impulsos neuronales con plásmidos (NP P Systems para abreviar), son un modelo computacional basado en el intercambio de pásmidos entre bacterias y fueron presentados en [1] por Francis George C Cabarle, Xiangxiang Zeng, Niall Murphy, Tao Song, Alfonso Rodríguez Patón y Xiangrong Liu. Se define un NP P System de grado $m \geq 1$ como una construcción de la forma:

$$\Pi = (O, b_1, ..., b_m, link, in, out)$$

donde:

a) $O = \{p_1, p_2, ..., p_w\}$ es un alfabeto no vacío de plásmidos.

b) $b_1, ..., b_m$ son bacterias de la forma $b_i = (N_i, R_i)$ para $1 \leq i \leq m$, donde:

- $N_i = \langle n_1, n_2, ..., n_w \rangle$ es un vector tal que $n_j \geq 0$ es el número inicial de plásmidos en la bacteria b_j para $1 \leq j \leq w$.
- R_i es un conjunto finito no vacío de reglas de la forma $C/r_1, r_2, ..., r_n$, donde $C \in O^+$ y toda r_k para $1 \leq k \leq n$ tiene una de las formas:
  - Operación de transmisión: $P \rightarrow out$, donde $P \in O^+$.
  - Operación de destrucción: $P \rightarrow \lambda$, donde $P \in O^+$.

c) $link \subset \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ con $(i, i) \notin link$ para todo $1 \leq i \leq m$.

d) *in* y *out* indican la bacteria de entrada y de salida respectivamente.

El funcionamiento del sistema es muy sencillo, en cada instante de tiempo se ejecuta una regla en cada bacteria (siempre que sea posible), esta regla se escogerá de manera indeterminista de entre las reglas aplicables.

Una regla $C/r_1, r_2, ..., r_n$ en la bacteria b_i se considera aplicable si al menos cada plásmido existente en C existe a su vez en b_i.

Cuando se ejecuta una regla, todas sus operaciones se ejecutan secuencialmente empezando por r_1 hasta r_n, las operaciones de transmisión ($P \rightarrow out$) envían una copia de cada plásmido en P a toda bacteria b_j tal que $(b_i, b_j) \in link$, por otro lado las operaciones de destrucción ($P \rightarrow \lambda$) eliminan los plásmidos de P en b_i (Es importante notar que las operaciones de transmisión no destruyen al plásmido en la bacteria original).

La computación ha acabado cuando ninguna bacteria puede aplicar una regla.

7.4.2 Simulación

Ahora que entendemos el funcionamiento de los NP P Systems podemos definir un procedimiento para convertir cualquier NP P Systems en un SNP-MC System con múltiples spikes, un posible procedimiento sería el siguiente:

a) Definimos tres spikes auxiliares a_p, s_p y h_p.

b) Creamos tres canales c_h, c_p y c_c.

c) Creamos la neurona *halt* con las regla $h_p^m \to h_p(c_h)$ y $h_p \to \lambda$.

d) Para cada $p_i \in O$ definimos un spike p_i.

e) Para cada bacteria $b_i = (N_i, R_i)(1 \leq i \leq m)$:

   (a) Creamos una neurona b_i, una neurona c_{b_i} y g_{b_i} (las tres contendrán originalmente un spike a_p y c_{b_i} contendrá también un spike s_p).

   (b) Añadimos al canal c_c las sinapsis (b_i, c_{b_i}), (c_{b_i}, b_i), (c_{b_i}, g_{b_i}) y (g_{b_i}, c_{b_i}).

   (c) Añadimos al canal c_h la sinapsis $(c_{b_i}, halt)$.

   (d) Para todo $n_j \in N_i$ añadimos n_j spikes p_j a la neurona b_i.

   (e) Añadimos la regla $a_p \to a_p(c_c)$ a la neurona g_{b_i}.

   (f) Añadimos las reglas $a_p s_p \to a_p(c_c)$ y $a_p \to a_p(c_c), h_p(c_h)$ a la neurona c_{b_i}.

   (g) Para toda regla de la forma $C/r_1, r_2, ..., r_n$ perteneciente a R_i definimos una regla en la que:

   - La expresión regular se define como $c_1 c_2 ... c_n p_1^* p_2^* ... p_w^*$ donde $c_1 c_2 ... c_n$ es la concatenación de todos los plásmidos de C.
   - El multiconjunto de spikes que consume la regla es la suma de todos los multiconjuntos de plásmidos consumidos por las operaciones de destrucción más el spike a_p.
   - Se envía el multiconjunto resultante de sumar todos los multiconjuntos de las operaciones de transmisión a través del canal c_p y el spike s_p a través del canal c_c.

   (h) Añadimos la regla $a_p \to \lambda$ a la neurona a_{b_i}.

f) Para cada $(i, j) \in link$ añadimos una sinapsis al canal c_p de la forma (b_i, b_j).

g) Se establece como neurona de entrada la correspondiente a la bacteria *in*.

h) Dada la bacteria de salida b_{out} añadimos al canal c_h las sinapsis $(halt, b_{out})$ y (b_{out}, out) añadimos a la neurona *out* las reglas $h_p p_1^* p_2^* ... p_w^* / p_i \to p_i(c_h)$ para todo $(1 \leq i \leq w)$ y $h_p \to \lambda$.

Las principales modificaciones necesarias para una correcta simulación del sistema han sido las siguientes:

- Hemos convertido las reglas del NP P System a reglas de nuestro sistema, para ello hemos tenido que añadir $p_1^* p_2^* ... p_w^*$ al final de la expresión regular para permitir que la regla se ejecute cuando se encuentran C plásmidos (spikes) o más.
- Hemos tenido que añadir el spike a_p que es consumido por todas las reglas para que una vez se lance una, no puedan activarse más.
- Para reponer el spike a_p en cada iteración está el sistema generador formado por las neuronas c_{b_i} y g_{b_i}, que envían un spike a_p en cada iteración a su respectiva bacteria (neurona) b_i.
- La última complicación nos la hemos encontrado con la bacteria de salida, pues en nuestro sistema solo se pueden enviar spikes a la salida, pero no recibir de ella ni aplicar reglas. Para solucionar esto, la bacteria de salida es una neurona, pero cuando en una bacteria no se aplica ninguna regla se envía un spike h_p a la neurona *halt*, si en un instante de tiempo dicha neurona recibe tantos spikes h_p como bacterias hay en el sistema querrá decir que la computación ha acabado, por lo que enviará una señal a la neurona de la bacteria de salida y esta volcará todo su contenido en el entorno.

7.5 Unificación de modelos

Como hemos visto a lo largo de esta sección, nuestro sistema es capaz de simular una amplia variedad de modelos de computación natural, lo que nos permite unificar estos modelos, ejecutar varios de manera paralela en el mismo sistema o incluso permitir que se comuniquen entre ellos transmitiéndose información.

7.6 Codificación de lenguajes

A la hora de codificar información podemos emplear variantes de los métodos descritos por José M Sempere en [11], donde se emplean distintas aproximaciones para codificar lenguajes con SNP Systems, de las cuales nos interesan las siguientes.

7.6.1 Ventana de tiempo

La primera aproximación consiste en leer la salida del sistema en grupos de k instantes de tiempo, este método es especialmente útil en el caso de los SNP Systems pues solo se puede enviar un spike a la salida en cada instante, por lo que para codificar más de dos símbolos es necesario agrupar la salida.

Posteriormente a esta salida se le aplica un mapeado de la forma $\varphi_k : B^k \to V_k$ donde k es el tamaño de la ventana, B^k es el conjunto de cadenas binarias de longitud k y V_k es el alfabeto destino.

Dado que nuestro modelo puede enviar al entorno no solo más de un spike por instante de tiempo, sino más de un tipo de spike, podemos emplear un tamaño de ventana $k = 1$ y nuestro mapeo pasaría a ser de la forma $\varphi : O^* \to V$ donde O^* es una cadena representando un multiconjunto sobre O y V es el alfabeto destino (notese que esta función no es inyectiva, pues existen distintas cadenas que codifican el mismo multiconjunto).

Llevando esta simplificación al extremo sería posible incluso prescindir del mapeo si $O = V$, en cuyo caso la codificación sería directa.

7.6.2 Codificación en cascada

La idea tras la codificación en cascada es emplear un conjunto finito de n SNP Systems (SNP MC Systems con múltiples símbolos en nuestro caso) conectados en tejido por un bus de datos, de este modo cuando el primer sistema alcanza un estado de parada envía su resultado como entrada del siguiente (pudiendo aplicar antes un mapeo φ). La salida del último sistema puede enviarse como entrada del primero si se desea una configuración iterativa.

Esto conecta directamente con los trabajos previos sobre computación con ADN y lenguajes formales de Vincenzo Manca, Carlos Martín Vide y Gheorghe Păun en [6] y de Vincenzo Manca en [5], donde se ha demostrado que las transducciones iterativas son capaces de caracterizar los lenguajes recursivamente enumerables.

7.7 Codificación multidimensional

A partir de los métodos de codificación vistos en el apartado anterior, podemos pasar a una codificación multidimensional de manera trivial si leemos la salida por cada canal de manera independiente (Lo que en la sección 3 hemos llamado *temporal multi-canal*). De este modo podemos aplicar cualquiera de las dos codificaciones anteriores pero a varios canales (o dimensiones) de manera simultánea.

8 Conclusiones y trabajos futuros

A lo largo de este trabajo hemos definido un nuevo modelo de computación natural, hemos diseñado un lenguaje de especificación, un intérprete y un simulador para poder experimentar de manera más cómoda con este nuevo modelo, y hemos obtenido una serie de resultados teóricos que demuestran la universalidad del modelo y su capacidad para simular otros sistemas de computación natural.

Esto nos abre la puerta a poder combinar estos sistemas y poder aprovechar sus capacidades y puntos fuertes combinados.

Queda mucho trabajo por delante, tanto en la parte teórica como en la de implementación, por lo que a continuación enumero algunos posibles trabajos futuros:

a) Añadir funcionalidad al lenguaje de especificación: funciones, lógica booleana, alias para poder trabajar más cómodamente con las membranas/neuronas...

b) Generalizar el lenguaje para poder definir otros sistemas que no sean SNP MC Systems con múltiples símbolos.

c) Compilar los modelos, actualmente los modelos se interpretan con un interprete construido en Python, lo ideal sería transpilarlos a C y posteriormente compilarlos para que su ejecución sea mucho más rápida.

d) Siguiendo en la línea del punto anterior, los modelos compilados podrían ejecutar cada neurona de manera realmente paralela para aprovechar las ventajas del modelo.

e) Y en el apartado teórico queda también mucho desarrollo, como estudiar las familias de lenguajes que es capaz de reconocer o generar el modelo en función del tamaño del alfabeto, del número de neuronas o del número de canales.

f) O como afecta cambiar el funcionamiento máximamente paralelo al sistema, variándolo por ejemplo a mínimamente paralelo.

9 Agradecimientos

Este trabajo no sería posible de no ser por la implicación del profesorado del máster, que se preocupa por incentivar al alumnado y animarle a aprender. Más concretamente, hubiera sido imposible llevarlo a cabo sin la ayuda de mi tutor José María Sempere Luna, profesor de la asignatura de Computación Natural sobre la que se sustenta principalmente este trabajo, que ha resuelto mis múltiples dudas y me ha ayudado a enfocar correctamente este TFM.

Por último pero no menos importante, agradecer a la fundación *ValgrAI – Valencian Graduate School and Research Network for Artificial Intelligence* asociación a la que estoy adscrito, por su apoyo económico mediante la ayuda ValgrAI para estudios de máster.

References

[1] Francis George C Cabarle, Xiangxiang Zeng, Niall Murphy, Tao Song, Alfonso Rodríguez-Patón, and Xiangrong Liu. Neural-like p systems with plasmids. *Information and Computation*, 281:104766, 2021.

[2] Xu Chen, Mario J Pérez-Jiménez, Luis Valencia-Cabrera, Beizhan Wang, and Xiangxiang Zeng. Computing with viruses. *Theoretical Computer Science*, 623:146–159, 2016.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *ACM SIGACT News*, 32:60–65, 3 2001.

[4] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural p systems. *Fundamenta informaticae*, 71(2-3):279–308, 2006.

[5] Vincenzo Manca. On the generative power of iterated transduction. In *Words, Semigroups, And Transductions: Festschrift in Honor of Gabriel Thierrin*, pages 315–327. World Scientific, 2001.

[6] Vincenzo Manca, Carlos Martın-Vide, and Gheorghe Păun. New computing paradigms suggested by dna computing: computing by carving. *BioSystems*, 52(1-3):47–54, 1999.

[7] Marvin Lee Minshy. *Computation: Finite and Infinite Machines*. Automatic Computation S. Prentice Hall, Old Tappan, NJ, December 1967.

[8] Robert Nystrom. *Crafting Interpreters*. Genever Benning, July 2021.

[9] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61, 2000.

[10] Hong Peng, Jinyu Yang, Jun Wang, Tao Wang, Zhang Sun, Xiaoxiao Song, Xiaohui Luo, and Xiangnian Huang. Spiking neural p systems with multiple channels. *Neural Networks*, 95:66–71, 2017.

[11] José M Sempere. Families of languages encoded by sn p systems. In *Membrane Computing: 18th International Conference, CMC 2017, Bradford, UK, July 25-28, 2017, Revised Selected Papers 18*, pages 262–269. Springer, 2018.