ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **14**, Number 2, 2011, 183–199

Computing Partial Recursive Functions by Deterministic Symport/Antiport P-Systems

Ionuţ DINCĂ

Department of Computer Science and Mathematics,
University of Piteşti
110040 Piteşti, Târgu din Vale, no.1, Romania
E-mail: `ionutdinca_24@yahoo.com`

Abstract. In this paper, we prove that deterministic 1-membrane systems with symport/antiport rules are Turing-complete. Our proof is based on the notion of numerical partial recursive function computed by numerical programs of a language with simple instructions, defined by Martin Davis and Elaine Weyuker [10]. We reduced the task from partial recursive functions to numerical programs and then to symport/antiport P systems.

1. Introduction

Membrane systems (or P Systems) were introduced by Gh. Păun (first time in the paper *Computing with membranes*, 2000 [7]) as formal computational models inspired by the structure and the functioning of living cells ([3, 8]). In the basic model, the computations are performed within a *membrane structure*, a rooted tree in which the nodes are membranes. *Multisets* of objects (sets of objects with multiplicities associated with the elements) are processed along the computations, making them to evolve and distributing them among the membranes. They evolve according to given *evolution rules*. The rules are applied in a nondeterministic and maximally parallel manner. In this way, one gets *transitions* from one configuration of the system to the next one. A maximal sequence of transitions constitutes a *computation*; a computation which reaches a configuration where no rule is applicable to the existing objects is a *halting computation*. The result of a halting computation is the number of objects collected in a specified output membrane. Many versions of this basic model were proposed for investigation. A comprehensive bibliography of membrane computing can be found at [11].

In this paper, we consider a variant called *symport/antiport P Systems with input membrane* which are very close to the basic model of symport/antiport P systems (first introduced in [9]). We prove that deterministic 1-membrane systems with symport/antiport rules and input membrane are Turing-complete. Our demonstration is based on the notion of numerical partial recursive function computed by numerical programs of a language with simple instructions, defined by Martin Davis and Elaine Weyuker [10]. This approach is not only more natural in terms of computer scientists but also provides clues about how classical programs written in the assembly languages of the processors based on silicon could be directly translated to hypothetical membrane system processors. In addition, we take the advantage of the reader's background in programming by developing computability theory in the context of an extremely simple abstract programming language proposed in [10].

We define what means that a partial numerical function is computed by such systems and provide a methodology for constructing a symport/antiport P system with input membrane for computing partial recursive functions. For that, we present a method for computing any partial computable function which is computed by some program. Since the class of partial recursive functions can be computed by numerical programs, we can compute any partial recursive function by deterministic 1-membrane systems with symport/antiport rules and input membrane.

2. Prerequisites

In this section, we present the notion of *multiset*, the main data structure of P systems, and the notion of *numerical program*, slightly modified from the original version in [10]: each instruction is uniquely identified by a label.

2.1. Multisets

Let U be an arbitrary set. A *multiset* (over U) is a mapping $M : U \longrightarrow \mathbf{N}$; $M(a)$, for $a \in U$, is the *multiplicity of a in the multiset M*. We denote by $M(U)$ the set of all multisets over U. If the set U is finite, $U = \{a_1, \ldots, a_n\}$, then the multiset M can be explicitly given in the form $\{(a_1, M(a_1)), \ldots, (a_n, M(a_n))\}$. The *support* of a multiset M is the set $supp(M) = \{a \in U \mid M(a) > 0\}$. A multiset M is empty when its support is empty (it is then denoted by $\emptyset$).

Let $M_1, M_2 : U \longrightarrow \mathbf{N}$ be two multisets. We say that M_1 is included in M_2, and we write $M_1 \subseteq M_2$, if $M_1(a) \leq M_2(a)$, for all $a \in U$. The inclusion is strict, and we write $M_1 \subset M_2$, if $M_1 \subseteq M_2$ and $M_1 \neq M_2$. The union of M_1 and M_2 is the multiset $M_1 \cup M_2 : U \longrightarrow \mathbf{N}$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$, for all $a \in U$. The difference $M_1 - M_2$ is defined here only when M_2 is included in M_1 and it is the multiset $M_1 - M_2 : U \longrightarrow \mathbf{N}$ given by $(M_1 - M_2)(a) = M_1(a) - M_2(a)$, for all $a \in U$.

A multiset M of finite support, $\{(a_1, M(a_1)), \ldots, (a_n, M(a_n))\}$, can also be represented by a string: $w = a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)}$ and all permutations of this string precisely identify the objects in the support of M and their multiplicities. If $M(a_i) = 0$ then the term $a_i^{M(a_i)}$ will be omitted. More details about the mathematics of multisets can be found in [5].

2.2. Numerical programs

As stated above, with respect to computability, we adopt the point of view from [10], developing computability approaches based on a simple abstract language $\mathcal{L}$ in order to take advantage of the reader's programming experience.

$$\begin{array}{c} \hline [L_1]I_1 \\ [L_2]I_2 \\ \ldots \\ [L_n]I_n \\ [L_{n+1}] \\ \hline \end{array}$$

Fig. 1. A program $\mathcal{P}$ of length n.

A program $\mathcal{P}$ of length n from $\mathcal{L}$ is a set of labelled instructions (all the instructions are labelled on the left and the labels of one program are different to each other) as in Fig. 1 where $I_i, 1 \leq i \leq n$, can be one of the following:
– $V \leftarrow V + 1$: increases by one the value of the variable V,
– $V \leftarrow V - 1$: if the value of V is zero, leave it unchanged; otherwise decrease by one the value of V,
– IF $V \neq 0$ GOTO L: if the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the sequence. The label L must be one of the labels that appear on the left of the programme.
In the above list V stands for any variable from:

– input variables $X_1, X_2, ...$, are used to store the input of the program;

– local variables $Z_1, Z_2, ...,$, initially zero, are used in the process of computation;

– output variable Y, initially zero, is used for the output of the program.

The label L stands for any label from $\{L_1, ..., L_n, L_{n+1}\}$. The label L_{n+1} is used for halting the program execution and it would not be followed by any instruction in the syntactic description of the programme. The empty program will be simply denoted by $[L_1]$ (the set of instructions is empty).
The data type of variables is the set of natural numbers $\mathbf{N}$.
Formally, the *state* of a program $\mathcal{P}$ is a map σ from the set of all variables $\{Y, X_1, X_2, X_3, ..., Z_1, Z_2, Z_3, ...\}$ into the set of natural numbers $\mathbf{N}$, only a finite number of variables having a natural number strictly grater than zero assigned to it. If σ is a state of $\mathcal{P}$ and V is a variable, then we denote by $\sigma(V)$ the value assigned to variable V by the state σ. A *snapshot* of a program of length n is a pair (L, σ), where $L \in \{L_1, ..., L_n, L_{n+1}\}$ and σ is a state (Intuitively, the label L indicates the next

instruction to be executed in the current state σ; $L = L_{n+1}$ corresponds to a "stop" instruction, and in that case the snapshot is called *terminal*).

We define the *successor* of the nonterminal snapshot $(L_i, \sigma), 1 \leq i \leq n$, to be the snapshot (L, τ) defined as follows:

Case 1. The instruction with label L_i is $V \leftarrow V + 1$ and $\sigma(V) = m$. Then $L = L_{i+1}$, $\tau(V) = m + 1$ and $\tau(V') = \sigma(V')$ for each variable V' other than V.

Case 2. The instruction with label L_i is $V \leftarrow V - 1$ and $\sigma(V) = m$. Then $L = L_{i+1}$ and if $m \neq 0$ then $\tau(V) = m - 1$ and $\tau(V') = \sigma(V')$ for each variable V' other than V; if $m = 0$, then $\tau = \sigma$.

Case 3. The instruction with label L_i is IF $V \neq 0$ GOTO A. Then $\tau = \sigma$ and there are two subcases:

Case 3a. If $\sigma(V) = 0$ then $L = L_{i+1}$.

Case 3b. If $\sigma(V) \neq 0$ then $L = A$.

In what follows, we define what means that a partial function of m variables is *computed* by a program $\mathcal{P}$.

Let $\mathcal{P}$ be any program in the language $\mathcal{L}$. For each $m \geq 1$ and each $m-$tuple $(r_1, ..., r_m)$ of natural numbers, the *initial snapshot* of the program $\mathcal{P}$ associated with the input $(r_1, ..., r_m)$ will be (L_1, σ), where σ is the state:

$$\sigma(X_1) = r_1, ..., \sigma(X_m) = r_m, \sigma(X_i) = 0, \text{ (for each } i > m),$$

and

$$\sigma(Y) = 0, \sigma(Z_i) = 0(\text{ for each } i > 0).$$

For each $m \geq 1$ and each input values $(r_1, ..., r_m)$, one of the following two cases exist:

Case 1. There is a finite sequence $s_1, s_2, ..., s_k$ of snapshots beginning with the initial snapshot s_1 associated with $(r_1, ..., r_m)$ (a *halting computation*) such that s_{i+1} is the successor of s_i for $i = 1, 2, ..., k - 1$ and the label from s_k is L_{n+1} (s_k is a *terminal* snapshot). The *result* of a halting computation is the value of variable Y in the corresponding terminal snapshot and we denote it by $\Psi_{\mathcal{P}}^{(m)}(r_1, ..., r_m)$.

Case 2. There is an *infinite* sequence $s_1, s_2, ...$ of snapshots beginning with the initial snapshot s_1 associated with $(r_1, ..., r_m)$ such that s_{i+1} is the successor of s_i. In this case, $\Psi_{\mathcal{P}}^{(m)}(r_1, ..., r_m)$ is *undefined.*

In this way, for each $m \geq 1$, a program $\mathcal{P}$ computes the partial function with m variables $\Psi_{\mathcal{P}}^{(m)}(x_1, ..., x_m)$.

A partial function $f : \mathbf{N}^m - \rightarrow \mathbf{N}$ is said that it is computed by some program $\mathcal{P}$ if

$$f(r_1, ..., r_m) = \Psi_{\mathcal{P}}^{(m)}(r_1, ..., r_m)$$

for all $(r_1, ..., r_m)$. Here this equation must be understood to mean not only that both sides have the same value when they are defined, but also that when either side of equation is undefined, the other is also. We say that f is a *partial computable function.*

Then, we need to construct more complex programs in our numerical language $\mathcal{L}$. For that, we introduce an abbreviating pseudo-instruction that will be called a *macro.* The program segment which it abbreviates will be called a *macro expansion.*

First, we need the macro $V \leftarrow 0$ which will have the effect of setting the content of V equal to 0. The macro expansion is simply

```
[$L_1$] $V \leftarrow V - 1$
[$L_2$] IF $V \neq 0$ GOTO $L_1$
[$L_3$]
```

Second, we need a macro $V \leftarrow U$ for the program which copies the value of U in the variable V while leaving the value of U unchanged. The macro expansion is

```
[$L_1$]$V \leftarrow 0$
[$L_2$]IF $U \neq 0$ GOTO $L_4$
[$L_3$]$Z_2 \leftarrow Z_2 + 1$
[$L_4$]IF $Z_2 \neq 0$ GOTO $L_{10}$
[$L_5$]$U \leftarrow U - 1$
[$L_6$]$V \leftarrow V + 1$
[$L_7$]$Z \leftarrow Z + 1$
[$L_8$]$Z_2 \leftarrow Z_2 + 1$
[$L_9$]IF $Z_2 \neq 0$ GOTO $L_2$
[$L_{10}$]IF $Z \neq 0$ GOTO $L_{13}$
[$L_{11}$]$Z_2 \leftarrow Z_2 + 1$
[$L_{12}$]IF $Z_2 \neq 0$ GOTO $L_{17}$
[$L_{13}$]$Z \leftarrow Z - 1$
[$L_{14}$]$U \leftarrow U + 1$
[$L_{15}$]$Z_2 \leftarrow Z_2 + 1$
[$L_{16}$]IF $Z_2 \neq 0$ GOTO $L_{10}$
[$L_{17}$]
```

where GOTO L_k is the macro for "unconditional branch" with the expansion

```
[$L_1$]$Z \leftarrow Z + 1$
[$L_2$] IF $Z \neq 0$ GOTO $L_k$
[$L_3$]
```

Then, let be a program $\mathcal{P}$ of length n. We will assume that the variables that occur in $\mathcal{P}$ are all included in the list $Y, X_1, ..., X_m, Z_1, ..., Z_p$. We write

$$\mathcal{P} = \mathcal{P}^{(n)}(Y, X_1, ..., X_m, Z_1, ..., Z_p)$$

in order that we can represent programs obtained from $\mathcal{P}$ by replacing the variables and labels by others. In particular, we will write

$$\mathcal{Q}_k = \mathcal{P}^{(n)}(Z_k, Z_{k+1}, ..., Z_{k+m}, Z_{k+m+1}, ..., Z_{k+m+p})$$

for each given value of k. The labels are also replaced in a corresponding manner (L_i is replaced by L_{k+i},$1 \leq i \leq n+1$).

Let $f(x_1, ..., x_m)$ be a partially computable function computed by some program $\mathcal{P} = \mathcal{P}^{(n)}(Y, X_1, ..., X_m, Z_1, ..., Z_p)$. We want to be able to use macros like

$$W \leftarrow f(V_1, ..., V_m)$$

in our programs, where $V_1, ..., V_m, W$ can be any variables. Such a macro will be an abbreviation of the following expansion:

```
$[L_1]Z_k \leftarrow 0$
$[L_2]Z_{k+1} \leftarrow V_1$
...
$[L_{m+1}]Z_{k+m} \leftarrow V_m$
$[L_{m+2}]Z_{k+m+1} \leftarrow 0$
...
$[L_{m+p+1}]Z_{k+m+p} \leftarrow 0$
$[L_{m+p+2}]\mathcal{Q}_k$
$[L_{m+p+3}]W \leftarrow Z_k$
```

The number k is chosen so large that none of the variables or labels used in $\mathcal{Q}_k$ occur in the main program of which the expansion is a part. When all the macros are expanded in a main program, the instructions are re-labelled in a corresponding manner.

Note 2.1. When we expand such simple macros in some main program, we have to replace all local variables and labels in a corresponding manner.

3. Partial Recursive Functions

The purpose of this section is to point out that using numerical programs it is possible to reproduce the behavior of any partial recursive function. Indeed, we are going to design programs such that: (i) compute *the basic or initial functions*: constant *zero* function, *successor* function and *projection* functions; (ii) compute the *composition* of functions, from programs computing the functions to compose; (iii) compute the *primitive recursion* of functions, from a program computing the function to iterate; (iv) compute the *unbounded minimization* of functions, from a program computing the function to minimize.

Computing Initial Functions.

We will design programs that allows us to compute the initial functions: constant zero function, successor function and projection functions.

- The *constant zero function*, $\mathcal{O} : \mathbf{N} \to \mathbf{N}$, is defined by $\mathcal{O}(r) = 0$, for every $r \in \mathbf{N}$. Clearly, this function can be computed by the empty program $[L_1]$.
- The *successor function*, $\mathcal{S} : \mathbf{N} \to \mathbf{N}$, is defined by $\mathcal{S}(r) = r+1$, for every $r \in \mathbf{N}$. This function can be computed by the program $\mathcal{P}_{\mathcal{S}}^{(7)}$:

```
[$L_1$]IF $X \neq 0$ GOTO $L_4$
[$L_2$]$Z \leftarrow Z + 1$
[$L_3$]IF $Z \neq 0$ GOTO $L_7$
[$L_4$]$X \leftarrow X - 1$
[$L_5$]$Y \leftarrow Y + 1$
[$L_6$]IF $X \neq 0$ GOTO $L_4$
[$L_7$]$Y \leftarrow Y + 1$
[$L_8$]
```

- The *projection functions*, $\mathcal{U}_j^n : \mathbf{N}^n \to \mathbf{N}$, with $n \geq 1$ and $1 \leq j \leq n$, are defined by $\mathcal{U}_j^n(r_1, ..., r_n) = r_j$, for every $(r_1, ..., r_n) \in \mathbf{N}^n$. A projection function can be computed by the program $\mathcal{P}_{\mathcal{U}_j^n}^{(6)}$:

```
[$L_1$]IF $X_j \neq 0$ GOTO $L_4$
[$L_2$]$Z \leftarrow Z + 1$
[$L_3$]IF $Z \neq 0$ GOTO $L_7$
[$L_4$]$X_j \leftarrow X_j - 1$
[$L_5$]$Y \leftarrow Y + 1$
[$L_6$]IF $X_j \neq 0$ GOTO $L_4$
[$L_7$]
```

Composition of Numerical Functions.

Definition 3.1. Let f be a function of k variables and let g_1,..., g_k be functions of m variables. Let

$$h(x_1, ..., x_m) = f(g_1(x_1, ..., x_m), ..., g_k(x_1, ..., x_m)).$$

Then h is said to be obtained from f and g_1,...,g_k *by composition.*

Let be $\mathcal{P}_{g_i}^{(n_i)}, 1 \leq i \leq k$, some programs which compute the functions $g_i, 1 \leq i \leq k$ respectively and some program $\mathcal{P}_f^{(n_f)}$ that computes the function f. Let us denote by $n'_i, 1 \leq i \leq k$, the lengths of the macro expansions of macros $Z_i \leftarrow g_i(X_1, ..., X_m), 1 \leq i \leq k$, respectively and by n'_f the length of the macro expansion of macro $Y \leftarrow f(Z_1, ..., Z_k)$.

A program which computes h could be the following:

```
$[L_1]$                                    $Z_1 \leftarrow g_1(X_1,...,X_m)$
$[L_{n'_1+1}]$                             $Z_2 \leftarrow g_2(X_1,...,X_m)$
...
$[L_{n'_1+...+n'_{k-1}+1}]$                $Z_k \leftarrow g_k(X_1,...,X_m)$
$[L_{n'_1+...+n'_k+1}]$                    $Y \leftarrow f(Z_1,...,Z_k)$
$[L_{n'_1+...+n'_k+n'_f+1}]$
```

Primitive Recursion of Numerical Functions.

Definition 3.2. Let f and g be total functions of n and $n+2$ variables, respectively. Let be the function h of $n+1$ variables defined by

$$\begin{cases} h(x_1,...,x_n,0) = f(x_1,...,x_n), \\ h(x_1,...,x_n,t+1) = g(t,h(x_1,...,x_n,t),x_1,...,x_n). \end{cases}$$

Then h is said to be obtained from f and g *by primitive recursion.*

Let be $\mathcal{P}_f^{(n_f)}$ and $\mathcal{P}_g^{(n_g)}$ some programs that compute the functions f and g respectively.

Let be the identity function $id(x) = x, \forall x \in \mathbf{N}$. Clearly, the identity function can be computed by $\mathcal{P}_{\mathcal{U}_1^1}^{(6)}$ where $\mathcal{U}_1^1$ is the projection function with one variable. Let us denote by n'_f the length of the macro expansion of macro $Y \leftarrow g_i(X_1,...,X_n)$, by n'_{id} the length of the macro expansion of macro $Z_3 \leftarrow id(Y)$ and by n'_g the length of the macro expansion of macro $Y \leftarrow g(Z_2,Z_3,X_1,...,X_n)$.

A program $\mathcal{P}_h$ which computes the function h could be the following:

```
$[L_1]$                              $Y \leftarrow f(X_1,...,X_n)$
$[L_{n'_f+1}]$                       IF $X_{n+1} \neq 0$ GOTO $L_{n'_f+4}$
$[L_{n'_f+2}]$                       $Z_1 \leftarrow Z_1 + 1$
$[L_{n'_f+3}]$                       IF $Z_1 \neq 0$ GOTO $L_{n'_f+n'_{id}+n'_g+7}$
$[L_{n'_f+4}]$                       $Z_3 \leftarrow id(Y)$
$[L_{n'_f+n'_{id}+4}]$               $Y \leftarrow g(Z_2,Z_3,X_1,...,X_n)$
$[L_{n'_f+n'_{id}+n'_g+4}]$          $Z_2 \leftarrow Z_2 + 1$
$[L_{n'_f+n'_{id}+n'_g+5}]$          $X_{n+1} \leftarrow X_{n+1} - 1$
$[L_{n'_f+n'_{id}+n'_g+6}]$          IF $X_{n+1} \neq 0$ GOTO $L_{n'_f+4}$
$[L_{n'_f+n'_{id}+n'_g+7}]$.
```

Unbound Minimization.

Finally, we design a program which computes the unbound minimization from the program that computes the function to be minimized. For that, we present here the unbound minimization operation.

Definition 3.3. Let be a partial computable function $f : \mathbf{N}^{m+1} \to \mathbf{N}$. The *unbound minimization operation* applied to f is the function $Min(f) : \mathbf{N}^m \to \mathbf{N}$ given by

$$Min(f)(x_1,...,x_m) = \begin{cases} y_{x_1,...,x_m}, & \text{if } y_{x_1,...,x_m} \text{ exists} \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

for every $x_1, ..., x_m \in \mathbf{N}^m$, where

$$y_{x_1,...,x_m} = \min\{y \in \mathbf{N} | \forall z < y(f \text{ is defined over } (x_1, ..., x_m, z)) \wedge f(x_1, ..., x_m, y) = 0\}$$

Let be $\mathcal{P}_f^{(n)}$ some program that computes f. Let us denote by n' the length of the macro expansion of macro $Z_1 \leftarrow f(X_1, ..., X_m, Y)$. A program $\mathcal{P}_{Min(f)}$ that computes $Min(f)$ could be defined as

```
[$L_1$]        $Z_1 \leftarrow f(X_1, ..., X_m, Y)$
[$L_{n'+1}$]   IF $Z_1 \neq 0$ GOTO $L_{n'+5}$
[$L_{n'+2}$]   $Z_2 \leftarrow Z_2 + 1$
[$L_{n'+3}$]   IF $Z_2 \neq 0$ GOTO $L_{n'+8}$
[$L_{n'+5}$]   $Y \leftarrow Y + 1$
[$L_{n'+6}$]   $Z_2 \leftarrow Z_2 + 1$
[$L_{n'+7}$]   IF $Z_2 \neq 0$ GOTO $L_1$
[$L_{n'+8}$].
```

Since the class of partial recursive functions coincides with the least class that contains the basic functions and is closed under composition, iteration and unbounded minimization (see [6]), it is then guaranteed that it is possible to construct programs that compute any partial recursive function.

In the next section we introduce the notion of *symport/antiport P-System with input membrane* and define the notion of partial function computed by such models. We present a method of computing any partial computable function which it is computed by some program. Consequently, we can prove that any partial recursive function can be computed by deterministic 1-membrane systems with symport/antiport rules and input membrane.

4. Symport/antiport P Systems with input membrane

Inspired from the computation model defined in [2] for the transition P-systems, we define here a similar model of computation for symport/antiport P-systems models. This includes new concepts like input membrane and a distinguished symbol Y for the definition of the output.

Definition 4.1. A *Symport/antiport P-System of degree $p \geq 1$ with input membrane* is a construct

$$\Pi = (\Gamma, \Sigma, Y, \mu, M_1, \ldots, M_p, E, R_1, \ldots, R_p, i_0, im),$$

where:

- Γ is the working alphabet of the system;
- Σ is the input alphabet such that $\Sigma \subset \Gamma$ and $Y \in \Gamma \setminus \Sigma$;

- Y is a distinguished element in $\Gamma \setminus \Sigma$;
- μ is a membrane structure with p membranes (labeled by $1, 2, \dots, p$); usually, 1 labels the skin membrane;
- $M_1, \dots, M_p$ are strings over Γ representing the multisets of objects initially present in the regions of the system;
- $E \subseteq \Gamma$ is the alphabet of the environment (here the objects are presents in arbitrarily many copies);
- $R_1, \dots, R_p$ are finite sets of *rules* for regions $1, 2, \dots, p$, respectively; the form of a rule is $(x, out; y, in)$, for $x, y \in \Gamma^*$ with $xy \neq \lambda$; if one of x, y is empty, then we have a *symport* rule, when both x and y are non-empty we have an *antiport* rule; the symport rules are written in the form (x, out) or (y, in), indicating only the non-empty string;
- im is a membrane of μ, indicating the input membrane of the system;
- i_0 is a membrane of μ indicating the output region.

The semantics of a rule $(x, out; y, in)$ from R_i, $1 \leq i \leq p$, is as follows: the objects present in x exit the region of membrane i, and, simultaneously, the objects present in y enter the region of membrane i. Obviously, the successful application of the rule is only possible if the region contains x and its parent region contains y.

In a symport rule of the form (x, out) or (x, in), the objects from x exit, respectively enter the region of membrane i in the multiplicity given by x.

As usual, a sequence of transitions is called a computation, and with any halting computation we associate an output. The multiplicity of the symbol Y present in region i_0 in the halting configuration is said to be the number computed by the system along that computation (the output); because of the nondeterminism of the application of rules, starting from an initial configuration, we can get several successful computations, hence several results. The set of all numbers computed in this way by Π is denoted by $N(\Pi)$.

For a detailed presentation about symport/antiport P Systems see, for example, [3].

When defining the initial configurations of the system, we must take into account that the system has an input membrane.

Definition 4.2. There exists an initial configuration for each multiset $m \in M(\Sigma)$ that can be introduced in the input membrane, namely,

$$C^{(0)}(m) = (\mu, M_1, ..., M_{im} \cup m, ..., M_p),$$

where M_i, $i \in \{1, ..., p\}$ are the initial multisets.

Given two configurations, C and C', of Π, we say that C' is obtained from C in one step, and we write $C \Rightarrow_\Pi C'$, if we can pass from the first one to the second by using the evolution rules associated with the membranes appearing in the membrane structure of C in a parallel and maximal way. If no configuration can be derived from C by applying those evolution rules, then we say that it is a *halting configuration.*

Definition 4.3. A configuration of order k is a configuration

$$C^{(k)} = (\mu, M_1^{(k)}, ..., M_p^{(k)}),$$

which can be reached from an initial configuration in exactly k steps.

Definition 4.4. *A halting computation*, $\mathcal{C}$, is a finite sequence of configurations $\{C^{(i)}\}_{i\leq r}$, where

- $C^{(0)}$ is an initial configuration of the system;
- $C^{(i)} \Rightarrow_\Pi C^{(i+1)}$, for every $i < r$;
- $C^{(r)}$ is a halting configuration and $r \in \mathbf{N}$;

A non halting computation, $\mathcal{C}$, is an infinite sequence of configurations $\{C^{(i)}\}_{i\in\mathbf{N}}$, where:

- $C^{(0)}$ is an initial configuration of the system;
- $C^{(i)} \Rightarrow_\Pi C^{(i+1)}$, for every $i \in \mathbf{N}$.

4.1. Computing Numerical Functions

We will define here what means that a symport/antiport P systems with output and input membrane computes a partial function between natural numbers, $f : \mathbf{N}^m - \to \mathbf{N}$. From that, we will impose that the input alphabet is ordered. In this way, it makes sense to consider that the multisets received as input represent tuples of natural numbers.

Definition 4.5. *A computing P system*, $\Pi^{(m)}$, of the order m is a P system that verifies the following:

- Π is a symport/antiport P Systems with input membrane;
- The input alphabet, Σ, of Π is an ordered alphabet with m elements. We denote it by $\Sigma = \{a_1, ..., a_m\}$;
- The output of a computation $\mathcal{C} = \{C^{(i)}\}_{i\leq r}$ is given by the following function:

$$Output(\mathcal{C}) = \begin{cases} undefined, & if\ \mathcal{C} \text{ is not halting,} \\ M_{i_0}^{(r)}(Y), & if\ \mathcal{C} \text{ halts and performs } r \text{ steps.} \end{cases}$$

According to the function definition, given a tuple of natural numbers, either the function is undefined over that tuple, or it is defined and returns a single value. This does not happen for nondeterministic membrane systems: for the same input data there can exist computations that are halting and others that are not halting; furthermore, the output of two halting computations over the same input data do not have to be the same tuple. Therefore, we must impose that our systems capture these properties.

Definition 4.6. A computing symport/antiport P-System, $\Pi^{(m)}$, of order m is said to be *valid* if it verifies the following:

- $\Pi^{(m)}$ is a symport/antiport P Systems of order m;
- Given an initial configuration, C, of $\Pi^{(m)}$, either no computation with initial configuration C is halting, or every computation with initial configuration C is halting;
- If $\mathcal{C}_1$ and $\mathcal{C}_2$ are two halting computations of $\Pi^{(m)}$ with the same initial configuration, then $Output(\mathcal{C}_1) = Output(\mathcal{C}_1)$.

The computing systems with input membranes from the above definition allow us to compute partial functions between natural numbers, according to the following definition.

Definition 4.7. A symport/antiport P System with input membrane $\Pi^{(m)}$ of order m computes the partial function $f : \mathbf{N}^m - \rightarrow \mathbf{N}$ if the following conditions are verified for each $(k_1, ..., k_m) \in \mathbf{N}^m$:

- f is defined over $(k_1, ..., k_m)$ if and only if there exists a halting computation of Π with the multiset $a_1^{k_1}...a_m^{k_m}$ as input;
- if $\mathcal{C}$ is a halting computation of Π with the multiset $a_1^{k_1}...a_m^{k_m}$ as input, then $Output(\mathcal{C}) = f(k_1, ..., k_m)$.

Notation. We denote by $\Psi^{(m)}_{\Pi^{(m)}}(k_1, ..., k_m) = f(k_1, ..., k_m)$ the function computed by $\Pi^{(m)}$.

5. Computational Completeness by Computing Partial Recursive Functions

In this section, we present a method for constructing, for any partial computable function f with m variables computed by some program $\mathcal{P}$, a deterministic 1-membrane system of order m with symport/antiport rules $\Pi^{(m)}_{\mathcal{P}}$ which computes exactly the partial function with m variables $\Psi^{(m)}_{\mathcal{P}}$. In this way, $\Pi^{(m)}_{\mathcal{P}}$ actually computes the function f.

Since the class of partial recursive functions can be computed by numerical programs, we reduce the task of the partial recursive functions to the numerical programs and then to the deterministic 1-membrane symport/antiport P systems.

Theorem 5.1. *For any program $\mathcal{P}$ and each $m \geq 1$, there exists a deterministic 1-membrane system $\Pi_{\mathcal{P}}^{(m)}$ of order m with symport/antiport rules which computes exactly the partial function with m variables $\Psi_{\mathcal{P}}^{(m)}$.*

Proof. Let be a program $\mathcal{P}$ and $m \geq 1$. We denote by n the length of $\mathcal{P}$. We want to construct a deterministic 1-membrane system $\Pi_{\mathcal{P}}^{(m)}$ of order m with symport/antiport rules which computes $\Psi_{\mathcal{P}}^{(m)}$. Therefore, we need to prove that, for each input $r_1, ..., r_m$, the output of the computation from $\Pi_{\mathcal{P}}^{(m)}$ starting with the initial configuration associated to input multiset representing the tuple $r_1, ..., r_m$ is the same with the result of the corresponding computation in $\mathcal{P}$ beginning with the initial snapshot associated with $(r_1, ..., r_m)$ (which is exactly the value $\Psi_{\mathcal{P}}^{(m)}(r_1, ..., r_m)$).

We define

$$\Pi_{\mathcal{P}}^{(m)} = (\Gamma, \Sigma, Y, \mu, M, E, R, 1, 1),$$

where:

- $\Gamma = E = \{Y, X_1, ..., X_m\} \cup \{X_i | \ X_i$ appears in $\mathcal{P}, \ i > m\} \cup \{Z_i | \ Z_i$ appears in $\mathcal{P}\}$ $\cup \ \{L_k, L_k^{'}, L_k^{''}, L_k^{'''}, L_k^{iv} | 1 \leq k \leq n\} \cup \{L_{n+1}\}$,
- $\Sigma = \{X_1, ..., X_m\}$,
- $M = L_1$,
- The set of rules R is generated by the algorithm *ConstructRules* from Fig. 2 (this idea was already used many times in membrane computing for the case of simulating ADD and SUB instructions in register machines).

Algorithm *ConstructRules*

```
Input: The program $\mathcal{P}$
Output: Set of symport/antiport rules $R$
$R := \{(L_{n+1}, out)\}$
for $k = 1$ to $n$ do
    if $I_k$ is of the form $V \leftarrow V + 1$ then
        $R := R \cup \{(L_k, out; V L_{k+1}, in)\}$
    if $I_k$ is of the form $V \leftarrow V - 1$ then
        $R := R \cup \{(L_k, out; L_k^{'} L_k^{''}, in), (L_k^{'} V, out; L_k^{'''}, in),$
        $(L_k^{''}, out; L_k^{iv}, in), (L_k^{iv} L_k^{'''}, out; L_{k+1}, in),$
        $(L_k^{iv} L_k^{'}, out; L_{k+1}, in)\}$
    if $I_k$ is of the form IF $V \neq 0$ GOTO $L$ then
        $R := R \cup \{(L_k, out; L_k^{'} L_k^{''}, in), (L_k^{'} V, out; L_k^{'''} V, in),$
        $(L_k^{''}, out; L_k^{iv}, in), (L_k^{iv} L_k^{'''}, out; L, in), (L_k^{iv} L_k^{'}, out; L_{k+1}, in)\}$
endfor
```

Fig. 2. Algorithm for constructing the set of rules R.

Let be an input $(r_1, ..., r_m)$ of $\mathcal{P}$ and $s_0 = (L_1, \sigma^{(0)})$ the initial snapshot associated with $r_1, ..., r_m$ and the corresponding computation

$$\mathcal{S} : s_0 = (L_1, \sigma^{(0)}), s_1 = (L_{i_1}, \sigma^{(1)}), s_2 = (L_{i_2}, \sigma^{(2)}), ...$$

where $i_1, i_2, ... \in \{1, 2, ..., n + 1\}$.

The corresponding initial configuration of $\Pi_{\mathcal{P}}^{(m)}$ associated with the input multiset $X_1^{r_1}...X_m^{r_m}$ will be in this case

$$C^0(X_1^{r_1}...X_m^{r_m}) = ([_1]_1, M^{(0)}),$$

where $M^{(0)} = X_1^{r_1}...X_m^{r_m} L_1$. Let be

$$\mathcal{C} : \ C^{(0)} = ([_1]_1, M^{(0)}), C^{(1)} = ([_1]_1, M^{(1)}), C^{(2)} = ([_1]_1, M^{(2)}), ...$$

(where $M^{(k)}$ is the multiset at the step k) the computation associated with the input multiset $X_1^{r_1}...X_m^{r_m}$.

We need to prove that $\mathcal{C}$ from $\Pi_{\mathcal{P}}^{(m)}$ correctly simulates $\mathcal{S}$ from $\mathcal{P}$ for each input $(r_1, ..., r_m)$.

For that, we prove by induction that for each two consecutive snapshots $s_k = (L_{i_k}, \sigma^{(k)})$ and $s_{k+1} = (L_{i_{k+1}}, \sigma^{(k+1)})$ from $\mathcal{S}$ there exist two configurations $C^{(i)}$ and $C^{(j)}$, $0 \leq i < j$, with $C^{(i)} \overset{j-i}{\Longrightarrow}_{\Pi_{\mathcal{P}}^{(m)}} C^{(j)}$ ($C^{(j)}$ is derived from $C^{(i)}$ in exactly $j - i$ steps) such that: (i) $M^{(i)}(L_{i_k}) = 1$ (contains the symbol L_{i_k}) and it does not contain any other version of symbol "L"; (ii) $M^{(j)}(L_{i_{k+1}}) = 1$ (contains the symbol $L_{i_{k+1}}$) and it does not contain any other version of symbol "L"; (iii) if $j - i > 1$ then the configurations $C^{(i+1)}, ..., C^{(j-1)}$ contain only labels from the set $\{L_{i_k}^{'}, L_{i_k}^{''}, L_{i_k}^{'''}, L_{i_k}^{iv}\}$; (iv) the transformation of $\sigma^{(k)}$ into $\sigma^{(k+1)}$ is correctly simulated by the execution of the above $j - i$ steps.

The basis for the induction. Let be $s_0 = (L_1, \sigma^{(0)})$ the current snapshot (initial snapshot) and $s_1 = (L_{i_1}, \sigma^{(1)})$ the snapshot obtained from s_0 following the definition of the successor snapshot. The corresponding configuration of s_0 is $C^{(0)}(M^{(0)})$ (therefore $i = 0$) where $M^{(0)} = X_1^{r_1}...X_m^{r_m} L_1$. We have the following cases:

- the instruction with the label L_1 is $V \leftarrow V+1$: In this case, the rule $(L_1, out; V L_2, in)$ can be executed; the multiplicity of variable V will be increased by 1 ($\sigma^{(1)}(V) = \sigma^{(0)}(V)+1$) and the label L_2 of the next instruction to be executed is introduced in the multiset ($M^{(j)}(L_2) = 1$). The configuration $C^{(j)}$ is reached in one step ($L_{i_1} = L_2$ and $j = 1$). Therefore the conditions (i)-(iv) are satisfied.

- the instruction with the label L_1 is $V \leftarrow V - 1$: In this case, we have the following two subcases:

  - $V \neq 0$: First, the rule $(L_1, out; L_1^{'} L_1^{''}, in)$ is applied. Then, the rules $(L_1^{'} V, out; L_1^{'''}, in)$,$(L_1^{''}, out; L_1^{iv}, in)$ are applied in parallel and the value of V is decreased by 1 ($\sigma^{(1)}(V) = \sigma^{(0)}(V) - 1$). Next, the rule $(L_1^{iv} L_1^{'''}, out;$ $L_2, in)$ is applied and the label L_2 of the next instruction to be executed is added to the current multiset ($M^{(j)}(L_2) = 1$). The configuration $C^{(j)}$ is reached in exactly 3 steps ($L_{i_1} = L_2$ and $j = 3$). Therefore the conditions (i)-(iv) are satisfied.

- $V = 0$: First, the rule $(L_1, out; L_1^{'}L_1^{''}, in)$ is applied. Then, the rules $(L_1^{''}, out; L_1^{iv}, in)$,$(L_1^{iv}L_1^{'}, out; L_2, in)$ are sequentially applied and the label L_2 of the next instruction to be executed is added to the current multiset ($M^{(j)}(L_2) = 1$). The configuration $C^{(j)}$ is reached in exactly 2 steps ($L_{i_1} = L_2$ and $j = 2$). Therefore the conditions (i)-(iv) are satisfied.

- the instruction with the label L_1 is IF $V \neq 0$ GOTO L: In this case, if $V \neq 0$ then the rule $(L_1, out; L_1^{'}L_1^{''}, in)$ can be executed; then, the rules $(L_1^{'}V, out; L_1^{'''}V, in)$, $(L_1^{''}, out; L_1^{iv}, in)$ can be applied in parallel. Next, the rule $(L_1^{iv}L_1^{'''}, out; L, in)$ can be applied and the symbol L of the next instruction to be executed will be added to the current multiset ($M^{(j)}(L) = 1$). The configuration $C^{(j)}$ is reached in exactly 3 steps ($L_{i_1} = L$ and $j = 3$). Therefore the conditions (i)-(iv) are satisfied. If $V = 0$ then the rules $(L_1, out; L_1^{'}L_1^{''}, in)$, $(L_1^{''}, out; L_1^{iv}, in)$, $(L_1^{iv}L_1^{'}, out; L_2, in)$ can be sequentially applied and the symbol L_2 will be added to the current multiset ($M^{(j)}(L_2) = 1$). The configuration $C^{(j)}$ is reached in exactly 3 steps ($L_{i_1} = L_2$ and $j = 3$). Therefore the conditions (i)-(iv) are satisfied.

The induction step. We suppose that the above assertion is true for some $k \in \mathbf{N}$ and we prove it for $k + 1$. Let be the consecutive snapshots $s_{k+1} = (L_{i_{k+1}}, \sigma^{(k+1)})$ and $s_{k+2} = (L_{i_{k+2}}, \sigma^{(k+2)})$ from $\mathcal{S}$. From the induction hypothesis, there exists a configuration $C^{(j)}$ which contains the symbol $L_{i_{k+1}}$ (and no other version of symbol "L"). In the following, the proof is similar as for the basis case with the role of L_1 played now by $L_{i_{k+1}}$ and the role of L_2 by $L_{i_{k+2}}$.

If the computation of $\mathcal{P}$ is finite then the terminal snapshot will contain the label L_{n+1} and in the last step of $\mathcal{C}$ the rule (L_{n+1}, out) will be applied leading to completion of the computation of $\Pi_{\mathcal{P}}^{(m)}$.

We can easily see that from any configuration of any computation from $\Pi_{\mathcal{P}}^{(m)}$ the next configuration is unique (the system is deterministic).

Since, for each input $(r_1, ..., r_m)$, the associated computation $\mathcal{S}$ from $\mathcal{P}$ is correctly simulated by the corresponding computation $\mathcal{C}$ from $\Pi_{\mathcal{P}}^{(m)}$, we have

$$\Psi_{\mathcal{P}}^{(m)}(r_1, ..., r_m) = \Psi_{\Pi_{\mathcal{P}}^{(m)}}^{(m)}(r_1, ..., r_m)$$

for any $r_1, ..., r_m$.

Therefore the function computed by $\Pi_{\mathcal{P}}^{(m)}$ is exactly the partial function with m variables $\Psi_{\mathcal{P}}^{(m)}$ and the theorem is proved. □

Note 5.1.

- According to Definition 4.6, the 1-membrane system with symport/antiport rules $\Pi_{\mathcal{P}}^{(m)}$ is a valid computing symport/antiport P-system.

- If a variable X_k with $k \in \{1, 2, ..., m\}$ does not appear in $\mathcal{P}$ then the symbol X_k with the multiplicity r_k will be ignored by the computation of $\Pi_{\mathcal{P}}^{(m)}$ associated with the input multiset $X_1^{r_1}...X_m^{r_m}$ (no rule is specified for the symbol X_k). Therefore the simulation is correct.

- On the other hand, if for a variable $X_k, k > m$, which appears in $\mathcal{P}$ it is not specified an input value then the symbol X_k will not exist in the initial configuration associated with the input multiset $X_1^{r_1}...X_m^{r_m}$. Therefore the rules that contain the symbol X_k correctly simulate the corresponding instructions from $\mathcal{P}$ (in this case, we recall that the value of the variable X_k is 0 in the initial state associated with the input $r_1, ..., r_m$). Consequently, the system $\Pi_{\mathcal{P}}^{(m)}$ can computes all the functions $\Psi_{\mathcal{P}}^{(k)}$, $1 \leq k \leq m$.

6. Related Work

In [2], a variant of the basic transition P-System is used for computing numerical functions. The idea behind this model is to be able to compute functions without worrying about the content of the membrane structure used to do it, but only considering the objects collected in its environment. The authors have proved the RE power of this computation models by simulating the behavior of the partial recursive functions.

In [1] the author introduces a variant of P-Systems with active membranes for which the computational completeness is proved by computing the class of partial recursive functions. They have used a new operation called "subordination": a membrane can be entirely absorbed by another membrane, preserving its content intact.

In [4], with inspiration from the economy, a new kind of P systems is proposed, where numerical variables evolve, starting from initial values, by means of production functions and repartition protocols. The authors prove that non-deterministic systems of this type, using polynomial production functions, characterize the Turing computable sets of natural numbers.

7. Conclusions

We have proposed here a method for computing numerical functions by symport/antiport P Systems with input membrane. We developed deterministic 1-membrane systems with symport/antiport rules which simulates the class of partial recursive functions. For that, we have designed such systems for computing any partial recursive function which is computed by some numerical program. This approach is more natural in terms of computer scientists and it could provide clues about how classical programs written in the assembly languages of the processors based on silicon could be directly translated to hypothetical membrane system processors. As future work, we want to continue using other important results from the computability theory based on numerical programs.

References

[1] Atanasiu A., *Recursive Calculus With Membranes*, Fundamenta Informaticae, **49**(1), pp. 45–59, 2002.

[2] Romero-Jiménez A., Pérez-Jiménez M. J., *Computing Partial Recursive Functions by Transition P Systems*, WMC 2003, LNCS 2933, Springer-Verlag, pp. 320–340, 2004.

[3] Păun Gh., *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, Heidelberg, 2002.

[4] Păun Gh., Păun R., *Membrane Computing and Economics: Numerical P Systems*, Fundamenta Informaticae, **73**(1), pp. 213–227, 2006.

[5] Calude C.S., Păun Gh., Rozenberg G., Salomaa A., eds., *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View*, LNCS, 2235, Springer, Berlin, 2001.

[6] Cohen D.E., *Computability and Logic,* Ellis Horwood, 1987.

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

[8] Păun Gh., Rozenberg G., Salomaa A., eds., *The Oxford Handbook of Membrane Computing*, Oxford Univ. Press, 2010.

[9] Păun A., Păun Gh., *The power of communication: P systems with symport/antiport*, New Generation Computing, **20**(3), pp. 295–305, 2002.

[10] Davis M.D., Sigal R., Weyuker E.J., *Computability, Complexity, and Languages*, Academic Press, 1994.

[11] The P Systems web page `http://ppage.psystems.eu/`.