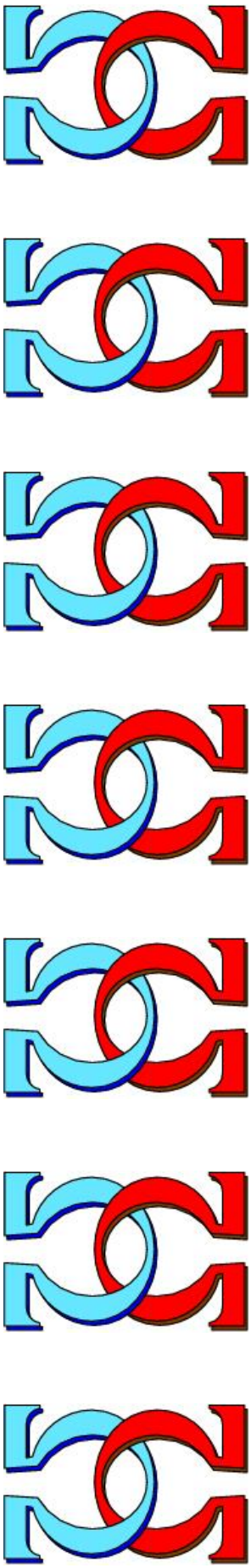

CDMTCS
Research
Report
Series

A New Universality Result on P Systems

Michael J. Dinneen
Yun-Bum Kim

Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-423
July 2012

Centre for Discrete Mathematics and
Theoretical Computer Science

A New Universality Result on P Systems

Michael J. Dinneen and Yun-Bum Kim

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand

{mjd,yun}@cs.auckland.ac.nz

Abstract

In this paper, we present a new universality result in P systems, by providing a method to construct a P system Π_M with two cells that simulates an arbitrary register machine M. The novelty of our approach is to utilize a very simple register machine that supports reading of binary data. In our simple construction of system Π_M: (i) cell states of a "simulator" cell in Π_M model the instruction lines of M, (ii) symbols (and their multiplicities) of Π_M model registers (and their values) of M, (iii) evolution rules of Π_M model the execution of instructions of M and (iv) communication to/from a "data" cell of Π_M is used to model the reading of data of M.

Keywords: P systems, register machines, universal computer.

1 Introduction

P systems (also called membrane systems) are distributed and parallel computing models, inspired by the structure and function of a living cell. A P system consists of a set of autonomous units, called membranes, that perform their own designated functions simultaneously. Several new P system models have been introduced, inspired from various features of living cells, that provide new ways to process information and solve the computational problems of interest.

Broadly speaking, as described by Nicolescu [Nic11], research on membrane systems falls into one of the following three areas: **(i) theory:** such as computational completeness (universality) [BG05, FKOS05, IPY06, CPPPJ07, CVMVV07, PP07], complexity classes (e.g. polynomial solutions to NP-hard problems [ZFM00, ILPW09]) or relationships with other models (e.g. automata, grammar systems and formal languages), **(ii) tools:** including designers, simulators and verifiers [GQGEPH$^+$09, GID09, GILD10, ILPH$^+$11], **(iii) applications:** such as computational biology, economics, ecosystem, linguistics and distributed computing [PP05, CCM$^+$11, II04, CDK02]. For a comprehensive list, we refer the readers to Păun et al.'s survey [PRS10].

Our research focuses on the practical applications of P systems in distributed computing—earlier we presented solutions, described at the P system level, to some distributed computing problems [DKN12, DKN10a, DKN10b].

In this paper, we present a new theoretical result, i.e. universality, in P systems. We provide the details of building a state-based P system Π that can simulate an arbitrary register machine M [CD07]. Specifically, we provide a set of evolution rules of system Π that simulates the instructions of register machine M. Note, the register machine that we consider here has input data bits, which are accessed using `READ` instructions that return the next unread bit. We contrast our approach with other universality results in P systems; their register machines need to store their input data (such as a register machine program to be simulated) in one of the registers [FKOS05, CVMVV07]. We have chosen this register machine model [CD07] mainly due to the inclusion of `READ` instructions. To our knowledge, the existing universality results, obtained by simulating a universal register machine, have not used machines with `READ` instructions—we provide P system evolution rules that can simulate these `READ` instructions.

The rest of this paper is organized as follows. Section 2 recalls the definitions of a state-based P system model, called a *simple P system*, and the register machine model of [CD07]. Section 3 presents the formal description of simple P systems that simulate any register machine (with binary input data). Finally, Section 4 summarizes this paper and provides some open problems.

2 Preliminaries

In this section, we recall the definitions of simple P systems and register machines used in this paper.

2.1 Simple P systems

A simple P system is defined as follows, which extends earlier versions of tissue and neural P systems [MVPPRP03, Pău02].

Definition 1. A *simple P system* of order n is a system $\Pi = (O, K, \Delta)$, where:

1. O is a finite non-empty alphabet of *symbols*.
2. $K = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is a finite set of *cells*, where each cell $\sigma_i \in K$ is of the form:

$$\sigma_i = (Q_i, s_{i0}, w_{i0}, R_i)$$

where

- Q_i is a finite set of *states*,
- $s_{i0} \in Q_i$ is the *initial state* ($s_i \in Q_i$ denotes the *current state*),
- $w_{i0} \in O^*$ is the *initial content* and ($w_i \in O^*$ denotes the *current content*),

- R_i is a finite *linearly ordered* set of evolution rules (i.e. transition multiset rewriting rules with *priority* and *rewrite operator*). An evolution rule $r \in R_i$ has the form:

$$r: \ j \ s \ u \rightarrow_\alpha s' \ v$$

where:

  - $\alpha \in \{\texttt{min}, \texttt{max}\}$ is a *rewriting* operator of r,
  - $j \in \mathbb{N}$ is the *priority* of r, where the lower value j indicates higher priority,
  - $s, s' \in Q_i$, where s' is the *target state* of r,
  - $u \in O^+$,
  - $v \in (O \times \tau)^*$, where a set of *target indicators* $\tau \in \{\odot, \uparrow, \downarrow, \updownarrow\}$. Note, $(o, \odot) \in v$, $o \in O$, is abbreviated to o.

The *initial configuration* of σ_i is denoted by (s_{i0}, w_{i0}) and the *current configuration* of σ_i is denoted by (s_i, w_i).

3. Δ is an irreflexive and asymmetric relation, representing a set of arcs between cells with *bidirectional* communication capabilities.

The rules are applied in the *weak priority* order [Pău06], i.e. (1) higher priority applicable rules are applied before lower priority applicable rules, and (2) a lower priority applicable rule is applied only if it indicates the same target state as the previously applied rules.

A cell *evolves* by applying one or more rules, which can change its content and state and can send objects to its neighbors. For a cell $\sigma_i = (Q_i, s_i, w_i, R_i)$, a rule $s \ x \rightarrow_\alpha s' \ x' \ (u)_\beta \mid z \in R_i$ is *applicable*, if $s = s_i$, $x \subseteq w_i$, $z \subseteq w_i$, $\Delta(i) \neq \emptyset$ for $\beta = \downarrow$, $\Delta^{-1}(i) \neq \emptyset$ for $\beta = \uparrow$ and $\Delta(i) \cup \Delta^{-1}(i) \neq \emptyset$ for $\beta = \updownarrow$.

The application of a rule transforms the current state s to the *target state* s', transforms multiset u to multiset $w = \bigcup\{o \mid (o, \odot) \in v\}$ and sends symbol x, where (x, τ'), $\tau' \in \{\uparrow, \downarrow, \updownarrow\}$, as specified by the transfer operator τ' (as further described below). Note that, multisets u and w will not be visible to other applicable rules in this same step, but they will be visible after all the applicable rules have been applied.

The rewriting operator $\alpha = \texttt{max}$ indicates that an applicable rewriting rule is applied as many times as possible. The rewriting operator $\alpha = \texttt{min}$ indicates that an applicable rewriting rule is applied once. For each application of a rule by cell σ_i, a copy of symbol o of $(o, \tau) \in v$ is replicated and sent to each cell $\sigma_j \in \Delta^{-1}(i)$ if $\tau = \uparrow$, $\sigma_j \in \Delta(i)$ if $\tau = \downarrow$ and $\sigma_j \in \Delta(i) \cup \Delta^{-1}(i)$ if $\tau = \updownarrow$, or remains in σ_i if $\tau = \odot$.

All applicable rules are applied in one *step*. An *execution* of a P system is a sequence of steps, that starts from the initial configuration. An execution *halts* if no further rules are applicable for all cells. The *computational results* of a halted system are the multiplicities of symbols present in the cells of the system.

We provide a simple P system example below. The purpose of this example is to demonstrate: (i) the structure of a system, (ii) the parallel processing power, between

cells and within each cell, and (iii) the manner in which evolution rules are assigned and applied.

Example 2. Consider a simple P system $\Pi = (\{a, b, c, d\}, \{\sigma_1, \sigma_2\}, \{(\sigma_1, \sigma_2)\})$, where each cell $\sigma_i \in K$ has the initial form $(\{s_0, s_1\}, s_0, w_{i0}, R)$, where:

- $w_{i0} = \begin{cases} a^3bc & \text{if } \sigma_i = \sigma_1, \\ a^6 & \text{if } \sigma_i = \sigma_2. \end{cases}$
- R is the following evolution rules:

  1 $s_0\ c \to_{\texttt{min}} s_0\ b\ (b, \downarrow)$

  2 $s_0\ b \to_{\texttt{max}} s_1$

  3 $s_0\ a \to_{\texttt{min}} s_0\ a\ a$

  4 $s_0\ a\ a \to_{\texttt{max}} s_1\ d$

The structure of system Π

The tables below illustrate the evolution of system Π. Columns "Initial state" and "Initial contents" indicate the current state and current contents of cell σ_i, $1 \leq i \leq 2$, respectively, prior to applying rules in the current step. Column "r_k", $1 \leq k \leq 4$, indicates the number of times each rule k has been applied in the current step. Columns "Final state" and "Final contents" indicate the resulting state and contents of cell σ_i, $1 \leq i \leq 2$, respectively, after applying the rules as indicated in columns "r_k", $1 \leq k \leq 4$.

	Cell σ_1								Cell σ_2							
Step	Initial state	Initial contents	r_1	r_2	r_3	r_4	Final state	Final contents	Initial state	Initial contents	r_1	r_2	r_3	r_4	Final state	Final contents
1	s_0	a^3bc	1	0	1	0	s_0	a^4b^2	s_0	a^6	0	0	1	0	s_0	a^7b
2	s_0	a^4b^2	0	2	0	2	s_1	d^2	s_0	a^7b	0	1	0	3	s_1	ad^3

We elaborate the importance of the destination states in the applications of evolution rules. As mentioned earlier, in each cell, all the rules assigned in a step must have the same destination state, which is set by the destination state of an applicable rule with the highest priority. For example, in cell σ_1, at step 1:

- An applicable rule with the highest priority, rule 1, can be assigned once. This assignment of rule 1 at step 1 sets the destination state as s_0, such that all other rules that will be assigned at step 1 must have the destination state s_0.
- The next rule in the priority, rule 2, has the destination state s_1, which is different from the destination state, s_0, set by rule 1. Hence, rule 2 cannot be assigned at step 1, even though rule 2 can be assigned once using the remaining unassigned symbols.
- The next rule in the priority, rule 3, has the destination state s_0 and rule 3 can be assigned once using the remaining unassigned symbols. Hence, rule 3 is assigned once together with rule 1 at step 1.

- The next rule in the priority, rule 4, has the destination state s_1. Hence, rule 4 cannot be assigned at step 1.

2.2 Register machines

A register machine, as presented in [CD07], has $n > 1$ instructions and $m > 0$ registers, where each register may contain an arbitrarily large non-negative integer. Register machines are Turing-complete and universal. A register machine program consists of a finite list of instructions, `EQ`, `SET`, `ADD`, `READ` and `HALT`, with the restriction that the `HALT` instruction appears only once, as the last instruction of the list, followed by an input data. The first instruction of a program is indexed by the value 0. In general, a program is presented in one of the following two forms: (i) symbolic instruction form and (ii) machine instruction (i.e. raw binary) form. For this paper, it suffices to use the symbolic form. We also note the motivating factor in designing the register machine model [CD07] was to have a very small (but practical) set of instructions, which also suits our purpose of minimizing the number of instruction cases to easily simulate by a P system. We note that this particular register machine language has evolved into a slightly more convenient syntax for establishing the "difficulty" complexity, based on smallest-known register machines, to decide/refute mathematical problems or conjectures [CCD06, CC10b, CC10a].

2.2.1 Instructions

A set of instructions of a register machine M, specified in [CD07] and denoted in Chaitin's style [Cha87], is listed below. In the instructions below, variables r_1, r_2 and r_3 denote registers and k denotes a non-negative binary integer constant.

1. Instruction: (`EQ` r_1 r_2 r_3) or (`EQ` r_1 k r_3)
   Assume that j denotes the content of r_3. If the content of r_1 equals (i) the content of r_2 or (ii) the constant k, then the execution of M continues at the j-th instruction. If the content of r_1 does not equal (i) the content of r_2 or (ii) the constant k, then the execution of M continues at the next instruction in the sequence.

2. Instruction: (`SET` r_1 r_2) or (`SET` r_1 k)
   The content of r_1 is replaced by (i) the content of r_2 or (ii) the constant k.

3. Instruction: (`ADD` r_1 r_2) or (`ADD` r_1 k)
   The content of r_1 is replaced by (i) the sum of the contents of r_1 and r_2 or (ii) the sum of the contents of r_1 and constant k.

4. Instruction: (`READ` r_1)
   One bit is read into r_1, so the numerical value of r_1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.

5. Instruction: (`HALT`)
   This is the last instruction of a register machine program.

2.2.2 Input data

In a register machine program, its input data, denoted as a sequence of bits (or characters), follows immediately after the halt instruction. Note, some programs may not have input data and it is up to the program to know how to process the data in the chosen encoding format.

Three examples of the possible ways to represent structured data are now given:

1. A simple but inefficient way to represent a non-negative integer n is by using a unary sequence of 0-bits of length n, where we terminate with a 1-bit. Example: 0001, 1 and 001 would represent the sequence of integers 3, 0 and 2.

2. To encode a sequence of integers, as a single integer, we require a leading 1-bit to be able to determine the number of 0-bits in the first element. Following [Din12] as an example: the array $[3, 0, 2]$ is represented by the integer $281_{(10)} = 100011001_{(2)}$.

3. A self-delimiting representation of a sequence of bits $b_1 b_2 \cdots b_k$, $b_i \in \{0, 1\}$, $1 \leq i \leq k$, is encoded as $1b_1 1b_2 \cdots 1b_k 0$. Note, if used to represent a positive integer n then we need only $O(k) = O(\lg n)$ bits, but twice the number of 'real' bits.

Later in this paper, we will use a variation of the last two described encodings to represent register machine data within a P system.

2.2.3 Run-time errors

A register machine program, with $n \geq 1$ instructions, can encounter the following run-time errors:

1. **Illegal branch error:** This error occurs when an `EQ` instruction is executed where the value indicated by its third register is greater or equal to n.

2. **Under-read error:** This error occurs if a register machine halts with unread input data, i.e. when the `HALT` instruction is encountered and there exist unread input data.

3. **Over-read error:** This error occurs if a register machine attempts to read past the last data-bit, i.e. when a `READ` instruction is encountered and the entire input data has already been read.

2.3 Register machine program example

The greatest common divisor (GCD) algorithm, based on subtraction, is as follows.

Algorithm 3. function `GCD`(x, y)

Input: $x \geq 0$ and $y \geq 0$.
Output: the final value of $GCD(x, y)$.
while $(x \neq 0)$
  if $(x < y)$ **then** `swap`(x, y)

$x := x - y$
end
return y

Example 4. We provide register machine instructions in symbolic form that correspond to the steps of the GCD Algorithm 3. Assume that, the input data is of form 0^x10^y1, which represents a sequence of two integers x and y. One register machine implementation, given next, has exactly 37 instructions where it reads any input values x and y from its data.

- Initialize registers $a, b, c, d, e, f, g, h, i$ with the instruction line numbers (to be used as targets in branching `EQ` instructions).

Line number	Symbolic instruction
0	`SET` a $\mathtt{L_a}$
1	`SET` b $\mathtt{L_b}$
2	`SET` c $\mathtt{L_c}$
3	`SET` d $\mathtt{L_d}$
4	`SET` e $\mathtt{L_e}$
5	`SET` f $\mathtt{L_f}$
6	`SET` g $\mathtt{L_g}$
7	`SET` h $\mathtt{L_h}$
8	`SET` i $\mathtt{L_i}$

- Initialize registers x and y, using auxiliary register z, with the first and the second values of the input data, respectively.

Line number	Symbolic instruction
9	$\mathtt{L_a}$: `READ` z
10	`EQ` z 1 b
11	`ADD` x 1
12	`EQ` a a a
13	$\mathtt{L_b}$: `READ` z
14	`EQ` z 1 c
15	`ADD` y 1
16	`EQ` b b b

- Check the condition $x = 0$.

Line number	Symbolic instruction
17	$\mathtt{L_c}$: `EQ` x 0 i

- Check the condition $x < y$.

Line number	Symbolic instruction
18	SET z x
19	SET w y
20	$\mathtt{L_d}$: EQ z y f
21	ADD z 1
22	ADD w 1
23	EQ x w f
24	EQ y z e
25	EQ d d d
26	$\mathtt{L_e}$: SET y x
27	SET x z

- Execute $x := x - y$.

Line number	Symbolic instruction
28	$\mathtt{L_f}$: SET z y
29	SET w 0
30	$\mathtt{L_g}$: EQ x z h
31	ADD z 1
32	ADD w 1
33	EQ g g g
34	$\mathtt{L_h}$: SET x w
35	EQ c c c

- Halt.

Line number	Symbolic instruction
36	$\mathtt{L_i}$: HALT

3 Universality results

Our preliminary results, presented at [Din12], have shown that simple P systems can simulate *non-data-based* register machines [CD07]. As mentioned earlier, one may try to use a special register to hold data and thus, with some effort (such as encoding an array of bits as an integer), develop a framework for showing P systems are universal. In this section, we extend our previous Turing completeness results by directly supporting the READ instruction to obtain a more straightforward universal result.

Given an arbitrary register machine M [CD07] with $n \geq 1$ instructions, $m \geq 0$ registers and input data bits $\beta = b_1 b_2 \cdots b_\nu$, we build a simple P system $\Pi_M = (O, K, \Delta)$, where:

1. $K = \{\sigma_m, \sigma_p\}$, where σ_m is called the *main* cell and σ_p is called the *provider* cell. The descriptions of these cells are given in Sections 3.1 and 3.2.

2. $\Delta = \{(\sigma_m, \sigma_p)\}$.

3. $O = \{r_i \mid 0 \leq i < k\} \cup \{\delta, \phi, \pi, \mu, \theta\}$, where symbols $r_0, r_1, \ldots, r_{k-1}$ represent the registers of M. Symbols π, γ and θ are auxiliary symbols, used only by the main cell, for executing the evolution rules that correspond to `HALT` and `EQ` instructions. Symbols μ, ϕ and δ are used as *communication messages*, i.e. requests and responses, between the main and provider cells during the execution the evolution rules that correspond to `READ` and `HALT` instructions—the manner in which these symbols are exchanged is described in Section 3.3.

3.1 The main cell

The role of the main cell is to simulate every instruction of M, except data-bit extraction—this operation is handled by the provider cell. The main cell, σ_m, is of the form $\sigma_m = (Q_m, s_{m0}, w_{m0}, R_m)$, where:

- $Q_m = \{s_i, s'_i \mid 0 \leq i < n\} \cup \{s\}$, where states s_i and s'_i, $0 \leq i < n$, represent the i-th instruction of M, state s_{n-1} represent the "halting" state and state s represents the "springboard jump" state. From state s, cell σ_m transits to state s_{j-1}, where $j \leq n$ is the multiplicity of symbol t that σ_m currently contains; if $j > n$, then σ_m remains at state s.

- $s_{m0} = s_0$, indicates the first instruction, i.e. 0-th instruction.

- $w_{m0} = \{r_i \mid 0 \leq i < k\} \cup \{\pi\}$, indicates the initial content of cell σ_m, where the value of each register r_i, $0 \leq i < k$, corresponds to the multiplicity of r_i minus one, i.e. $|w_m|_{r_i} - 1$.

- R_m corresponds to the instructions of M, described in the following subsections.

3.1.1 Evolution rules for a `SET` instruction

An i-th instruction of the form, either (`SET` r_{i_1} r_{i_2}) or (`SET` r_{i_1} k_i), is translated into the following evolution rules. The rules below first consume, all but one, copies of symbol r_{i_1} and then produce j additional copies of symbol r_{i_1}, where: (i) j is the multiplicity of symbol r_{i_2}, i.e. the value of the register r_{i_2}, or (ii) $j = k_i$, i.e. the value of constant k_i.

Instruction	Corresponding evolution rules
(`SET` r_{i_1} r_{i_2})	1 s_i r_{i_1} $\rightarrow_{\texttt{max}}$ s_{i+1}
	2 s_i r_{i_2} $\rightarrow_{\texttt{max}}$ s_{i+1} r_{i_1} r_{i_2}

Using the rules above, the main cell consumes all copies of symbol r_{i_1}, i.e. set the value of register r_{i_1} to 0. At the same time, the main cell rewrites every copy of symbol r_{i_2} into multiset $r_{i_1} r_{i_2}$, i.e. set the value of register r_{i_1} to the value of register r_{i_2}.

Instruction	Corresponding evolution rules
$(\texttt{SET}\ r_{i_1}\ k_i)$	$1\ s_i\ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}^{k_i+1}$
	$2\ s_i\ r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}$

Using the rules above, the main cell: (i) rewrites one copy of symbol r_{i_1} into $k_i + 1$ copies of symbol r_{i_1} and (ii) consumes all the remaining copies of symbol r_{i_1}, i.e. set the value of register r_{i_1} to constant k_i.

3.1.2 Evolution rules for an ADD instruction

An i-th instruction of the form, either $(\texttt{ADD}\ r_{i_1}\ r_{i_2})$ or $(\texttt{ADD}\ r_{i_1}\ k_i)$, is translated into the following evolution rules. The rules below produce j additional copies of symbol r_{i_1}, where: (i) j is the multiplicity of symbol r_{i_2}, i.e. the value of register r_{i_2} or (ii) $j = k_i$, i.e. the value of the constant k_i.

Instruction	Corresponding evolution rules
$(\texttt{ADD}\ r_{i_1}\ r_{i_2})$	$1\ s_i\ r_{i_2} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_2}$
	$2\ s_i\ r_{i_2} \rightarrow_{\texttt{max}} s_{i+1}\ r_{i_1}\ r_{i_2}$

Using the rules above, except one copy of symbol r_{i_1}, the main cell rewrites every copy of symbol r_{i_2} into multiset $r_{i_1} r_{i_2}$, i.e. set the value of register r_{i_1} to the sum of values of registers r_{i_1} and r_{i_2}.

Instruction	Corresponding evolution rule
$(\texttt{ADD}\ r_{i_1}\ k_i)$	$1\ s_i\ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}^{k_i+1}$

Using the rules above, the main cell rewrites one copy of symbol r_{i_1} into $k_i + 1$ copies of symbol r_{i_1}, i.e. set the value of register r_{i_1} to the sum of the value of register r_{i_1} and constant k_i.

3.1.3 Evolution rules for an EQ instruction

An i-th instruction of the form, $(\texttt{EQ}\ r_{i_1}\ r_{i_1}\ r_{i_3})$, $(\texttt{EQ}\ r_{i_1}\ r_{i_2}\ r_{i_3})$ or $(\texttt{EQ}\ r_{i_1}\ k_i\ r_{i_3})$, is translated into the following evolution rules. The multiplicity of symbol γ, produced by the rules below, corresponds to the difference in the values of: (i) registers r_{i_1} and r_{i_2} or (ii) register r_{i_1} and constant k_i. Let j denote the value of the register r_{i_3}. If the main cell does not contain any copies of symbol γ, then it produces j copies of symbol θ and transits to the "springboard" state, where the main cell's state transition will be determined according to the multiplicity of symbol θ. Otherwise, the main cell consumes all copies of symbol γ and transits to state s_{i+1}, i.e. the next instruction.

Instruction	Corresponding evolution rule
$(\texttt{EQ}\ r_{i_1}\ r_{i_1}\ r_{i_3})$	$1\ s_i\ r_{i_3} \rightarrow_{\texttt{max}} s\ r_{i_3}\ \theta$

Instruction	Corresponding evolution rules
$(\texttt{EQ}\ r_{i_1}\ r_{i_2}\ r_{i_3})$	**Rules for state s_i:**
	$1\ s_i\ r_{i_1}\ r_{i_2} \rightarrow_{\texttt{max}} s'_i\ r_{i_1}\ r_{i_2}$
	$2\ s_i\ r_{i_1} \rightarrow_{\texttt{max}} s'_i\ r_{i_1}\ \gamma$
	$3\ s_i\ r_{i_2} \rightarrow_{\texttt{max}} s'_i\ r_{i_2}\ \gamma$
	Rules for state s'_i:
	$1\ s'_i\ \gamma \rightarrow_{\texttt{max}} s_{i+1}$
	$2\ s'_i\ r_{i_3} \rightarrow_{\texttt{max}} s\ r_{i_3}\ \theta$

Instruction	Corresponding evolution rules
$(\texttt{EQ}\ r_{i_1}\ k_i\ r_{i_3})$	$1\ s_i\ r_{i_1}^{k_i+2} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}^{k_i+2}$
	$2\ s_i\ r_{i_1}^{k_i+1} \rightarrow_{\texttt{min}} s\ r_{i_1}^{k_i+1}$
	$3\ s_i\ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}$
	$4\ s_i\ r_{i_3} \rightarrow_{\texttt{max}} s\ r_{i_3}\ \theta$

The "springboard" state mimics the manner in which a register machine performs a "GOTO" operation to move to l-th instruction, $0 \leq l \leq n-1$. The springboard state contains one rule designated for each value $1, 2, \ldots, n$, such that $1 \leq j \leq n$ copies of symbol θ will lead the main cell to transit to state s_{j-1}. Additionally, the springboard state contains one extra rule designated for all values greater than n, such that $j > n$ copies of symbol θ will lead the main cell to infinite loop state transitions.

Rules for the "springboard" state s:

$1\ s\ \theta^{n+1} \rightarrow_{\texttt{min}} s\ \theta^{n+1}$

$2\ s\ \theta^{n} \rightarrow_{\texttt{min}} s_{n-1}$

$3\ s\ \theta^{n-1} \rightarrow_{\texttt{min}} s_{n-2}$

$\vdots$

$n+1\ s\ \theta \rightarrow_{\texttt{min}} s_0$

3.1.4 Evolution rules for a READ instruction

An i-th instruction of the form $(\texttt{READ}\ r_{i_1})$ is translated into the following evolution rules in cell σ_m. The rules below set the multiplicity of symbol r_{i_1} to either *one* or *two* (indicating a data-bit of value 0 or 1).

Setting the multiplicity of symbol r_{i_1} as described above requires interactions with the provider cell. Symbol μ represents the "last unread data-bit" request from the main cell to the provider cell. Symbols ϕ and δ represent the provider cell's responses, where (i) symbol ϕ indicates that the entire data has been read and (ii) symbol δ indicates that the last unread data-bit is 1. The provider cell's other response is not to send any symbol—this response indicates that the last unread data-bit is 0.

According to the provider cell's three possible responses, the main cell sets the multiplicity of symbol r_{i_1} as follows. Note, the main cell initially contains one copy of symbol

r_{i_1}. If the main cell receives: (i) symbol δ, then the main cell rewrites the received symbol δ into symbol r_{i_1}, such that the final multiplicity of symbol r_{i_1} is two, or (ii) symbol ϕ, then the main cell enters infinite loop state transitions. Otherwise, the main cell remains idle, such that the final multiplicity of symbol r_{i_1} remains at one.

Instruction	Corresponding evolution rules
(READ r_{i_1})	**Rules for state s_i:**
	1 $s_i\ r_{i_1} \rightarrow_{\texttt{min}} s'_i\ r_{i_1}\ (\mu, \downarrow)$
	2 $s_i\ r_{i_1} \rightarrow_{\texttt{max}} s'_i$
	Rules for state s'_i:
	1 $s'_i\ r_{i_1} \rightarrow_{\texttt{min}} s''_i\ r_{i_1}$
	Rules for state s''_i:
	1 $s''_i\ \phi \rightarrow_{\texttt{min}} s''_i\ \phi$
	2 $s''_i\ \delta \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}$
	3 $s''_i\ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}$

3.1.5 Evolution rules for a HALT instruction

The last instruction must be HALT, which is translated into the following evolution rules. According to the rules below, the main cell reaches either: (i) a halting configuration or (ii) an infinite loop configuration. The rules below involve the interaction between the main and provider cells, where the responses from the provider cell are as described in the READ instruction of Section 3.1.4. If the provider cell's response is symbol ϕ, then the main cell reaches a halting configuration. Otherwise, the main cell enters an infinite loop configuration.

Instruction	Corresponding evolution rules
(HALT)	**Rules for state s_{n-1}:**
	1 $s_{n-1}\ \pi \rightarrow_{\texttt{min}} s'_{n-1}\ \pi\ (\mu, \downarrow)\ (\mu, \downarrow)$
	Rules for state s'_{n-1}:
	1 $s'_{n-1}\ \pi \rightarrow_{\texttt{min}} s''_{n-1}\ \pi$
	Rules for state s''_{n-1}:
	1 $s''_{n-1}\ \phi\ \pi \rightarrow_{\texttt{min}} s_{n-1}$
	2 $s''_{n-1}\ \pi \rightarrow_{\texttt{min}} s''_{n-1}\ \pi$

3.2 The provider cell

The role of the provider cell is to obtain and send the last unread bit to the main cell. For an input data of ν bits, $\beta = b_1 b_2 \cdots b_\nu$, where $b_i \in \{0, 1\}$ and $1 \leq i \leq \nu$, the provider cell initially contains the multiset δ^k, where k is the value of the binary-encoded integer $\beta' = 1 b_\nu b_{\nu-1} \cdots b_1$. The bit of value 1 at the first position of β', which is not part of β,

ensures that if the last $k \geq 1$ bits of β, i.e. $b_{\nu-k}b_{\nu-k+1}\cdots b_{\nu}$, are of value 0, then we do not lose them.

Let $x > 1$ denote the current multiplicity of symbol δ in cell σ_p. At the i-th `READ` instruction, cell σ_p performs "`mod` 2" operation on value x to obtain the i-th bit. Then σ_p performs a "`div` 2" operation on the value of x to prepare for the next `READ` instruction, if any. For example, if σ_p has $x = 11_{(10)} = 1011_{(2)}$ copies of symbol δ, then the next three successive bits returned to the main cell σ_m are 1, 1 and 0.

The provider cell, σ_p, is of the form $(Q_p, s_{p0}, w_{p0}, R_p)$, where:

- $Q_p = \{s, s'\}$.
- $s_{p0} = s$.
- $w_{p0} = \{\delta^{\beta'}\}$, where $\beta' = 1b_nb_{n-1}\cdots b_{1(2)}$.
- R_p is the following rules, which correspond to the `READ` and `HALT` instructions of M.

Rules for state s:
$1\ s\ \delta\ \delta\ \mu \rightarrow_{\texttt{min}} s'\ \delta$
$2\ s\ \delta\ \mu \rightarrow_{\texttt{min}} s'\ \delta\ (\phi, \uparrow)$
$3\ s\ \delta \rightarrow_{\texttt{min}} s\ \delta$
$4\ s\ \delta\ \delta \rightarrow_{\texttt{max}} s'\ \delta$
$5\ s\ \delta \rightarrow_{\texttt{min}} s'\ (\delta, \uparrow)$
Rules for state s':
$1\ s'\ \delta\ \mu\ \rightarrow_{\texttt{min}} s$
$2\ s'\ \delta \rightarrow_{\texttt{min}} s\ \delta$

The provider cell receives at most two copies of symbol μ from the main cell in a single step. Using the first copy of symbol μ, if any, the provider cell performs the "`mod` 2" and "`div` 2" operations on the current multiplicity of symbol δ as described above. Using the second copy of symbol μ, if any, the provider cell reaches a halting configuration.

Let j denote the current multiplicity of symbol δ. The provider cell notifies the results of a "`mod` 2" operation on the value j to the main cell by: (i) not sending any symbol to indicate that j `mod` $2 = 0$ (ii) sending one copy of symbol δ to indicate that j `mod` $2 = 1$, or (iii) sending one copy of symbol ϕ to indicate that $j = 1$, i.e. the all data-bits $b_1b_2\cdots b_{\nu}$ have been read.

3.3 Handling of run-time errors

In Section 2.2, we described the run-time errors of a register machine program. We now describe the manner in which the cells of system Π_M detect and handle the run-time errors. Figure 1 illustrates the provider cell's three possible responses, upon receiving a data-bit request (i.e. symbol μ) from the main cell. From these responses, the main cell detects the over-read and under-read errors.

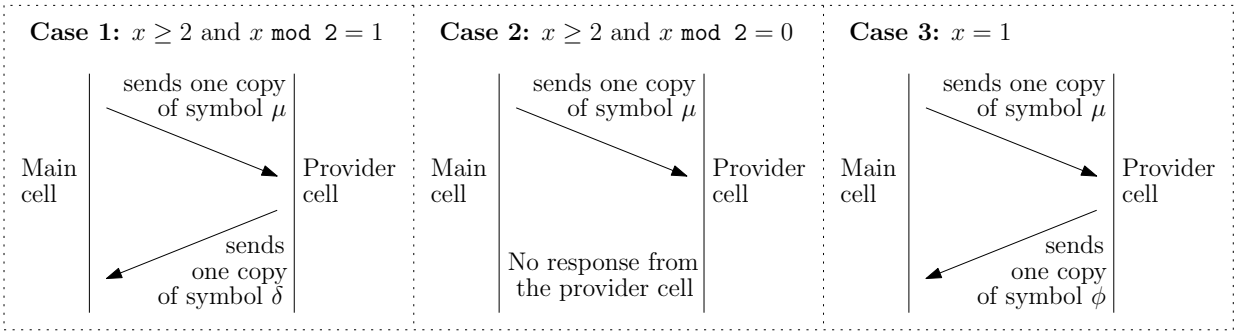

Figure 1: Symbol exchanges between the main and provider cells in the `READ` and `HALT` instructions, where x indicates the current multiplicity of symbol δ in the provider cell.

3.3.1 Over-read error

This error can be detected in a `READ` instruction. When the main cell encounters a `READ` instruction, it sends one copy of symbol μ to the provider cell. The purpose of this symbol μ is to request the next unread bit. The main cell interprets the provider cell's responses as follows.

- One copy of symbol δ indicates that the next bit is 1.
- "No response", i.e. case 2, indicates that the next bit is 0.
- One copy of symbol ϕ indicates that the entire input data has been read, i.e. an over-read error. In this case, the main cell enters an infinite loop, hence system Π_M does not halt.

3.3.2 Under-read error

This error can be detected in the `HALT` instruction. When the main cell encounters the `HALT` instruction, it sends one copy of symbol μ to the provider cell. The purpose of this symbol μ is to confirm that the entire input data have been read. The main cell interprets provider's responses as follows.

- One copy of symbol ϕ indicates that the entire input data has been read. In this case, both the main and provider cell do not apply rules any more, hence system Π_M halts.
- One copy of symbol δ or "no response" (i.e. case 2) indicates that there are unread input data, i.e. an under-read error. In this case, the main cell enters an infinite loop, hence system Π_M does not halt.

3.3.3 Illegal branching error

This error can be detected in an `EQ` instruction. If the values of the first registers (or the first register and the constant) of an `EQ` instruction are the same, then the main cell produces $j + 1$ copies of symbol θ, where j is the value of the third register of the `EQ`

instruction. Let n denote the number of instructions. If $j+1 \geq n$ then the value of the third register must be greater or equal to n. Thus, the main cell detects (in the "springboard" state s) an illegal branching error and enters an infinite loop. Hence, system Π_M does not halt.

3.4 Analysis of system Π_M and remarks

When system Π_M halts, the configuration of (i) the provider cell is $(s, \emptyset)$ and (ii) the main cell is (s_{n-1}, w_m), where $w_m = \{r_i \mid 0 \leq i < n\}^*$. The computational results of a halted system Π_M is the final content of the main cell, where the multiplicity of symbol r_i minus one, $0 \leq i < n$, indicates the value of the register r_i of register machine M.

Theorem 5. Simple P systems are universal.

Proof. In Section 3, we presented the details of building a simple P system Π_M that simulates any register machine M (with input data) [CD07]. Thus, simple P systems are universal. □

The number of evolution rules in system Π_M, which simulates a given machine M with n instructions, is proportional to n. The lower and upper bounds on the number of evolution rules of system Π_M are indicated in Proposition 6.

Proposition 6. For register machine M with n instructions, there are $n_{\texttt{SET}}$ `SET` instructions, $n_{\texttt{ADD}}$ `ADD` instructions, $n_{\texttt{EQ}}$ `EQ` instructions, $n_{\texttt{READ}}$ `READ` instructions and one `HALT` instruction, such that $n = n_{\texttt{EQ}} + n_{\texttt{SET}} + n_{\texttt{ADD}} + n_{\texttt{READ}} + 1$. The corresponding system Π_M contains $n_{\texttt{total}}$ evolution rules, where $n_{\texttt{EQ}}+2\cdot n_{\texttt{SET}}+n_{\texttt{ADD}}+6\cdot n_{\texttt{READ}}+(n_{\texttt{EQ}}/n_{\texttt{EQ}})\cdot(n+1)+12 \leq n_{\texttt{total}} \leq 5 \cdot n_{\texttt{EQ}} + 2 \cdot n_{\texttt{SET}} + 2 \cdot n_{\texttt{ADD}} + 6 \cdot n_{\texttt{READ}} + (n_{\texttt{EQ}}/n_{\texttt{EQ}}) \cdot (n+1) + 12$.

Proof. Recall the set of evolution rules given in Sections 3.1 and 3.2.

There are two evolution rules for each of instructions (`SET` r_{i_1} r_{i_2}) and (`SET` r_{i_1} k_i). Thus, for $n_{\texttt{SET}}$ number of `SET` instructions, there are $2 \cdot n_{\texttt{SET}}$ evolution rules.

There are two and one evolution rules for each of instructions (`ADD` r_{i_1} r_{i_2}) and (`ADD` r_{i_1} k_i), respectively. If all $n_{\texttt{ADD}}$ instructions are of type (`ADD` r_{i_1} r_{i_2}), then there are $n_{\texttt{ADD}}$ number of evolution rules. If all $n_{\texttt{ADD}}$ instructions are of type (`ADD` r_{i_1} k_i), then there are $2 \cdot n_{\texttt{ADD}}$ number of evolution rules.

There are one, five and four evolution rules for instructions (`EQ` r_{i_1} r_{i_1} r_{i_3}), (`EQ` r_{i_1} r_{i_2} r_{i_3}) and (`EQ` r_{i_1} k_i r_{i_3}), respectively, and $n+1$ additional evolution rules for the "springboard" state. If all $n_{\texttt{EQ}}$ instructions are of type (`EQ` r_{i_1} r_{i_1} r_{i_3}), then there are $n_{\texttt{EQ}} + n + 1$ number of evolution rules. If all $n_{\texttt{EQ}}$ instructions are of type (`EQ` r_{i_1} r_{i_2} r_{i_3}), then there are $5 \cdot n_{\texttt{EQ}} + n + 1$ number of evolution rules.

The main cell contains (i) six evolution rules for each `READ` instruction and (ii) four evolution rules for the `HALT` instruction. The provider cell contains seven evolution rules, which are associated with `READ` and `HALT` instructions. □

4 Conclusions

In this paper, we presented a new universality result in P systems. Specifically, we presented a state-based simple P system that can simulate any register machine of the

syntax given in [CD07]. In Section 3, we proved that simple P systems are universal by presenting the details of building a simple P system Π_M that simulates any register machine M (with input data) [CD07]. Each constructed system Π_M has the following properties: (i) there are two cells, (ii) the number of states and evolution rules are proportional to the number of instructions of a given register machine and (iii) the number of P system steps required for each register machine instruction during the simulation is constant.

The translation presented in this paper builds a simple P system with two cells. The purpose of having two cells is to designate specific functions to different cells, i.e. the provider cell handles IO operations on the input data. Additionally, the inclusion of the provider cell highlights the progress from our previous Turing completeness results on simple P systems—we proved the Turing completeness by providing the details of building a simple P system that simulates a register machine [CD07] *without* input data and `READ` instructions. As a possible future work, we could consider presenting a translation that builds a simple P system with one cell, i.e. merge the evolution rules of the main and provider cells into a single cell. An advantage of having one cell is that we can eliminate the communications between the main and provider cells, where each communication causes one step delay before receiving a response, such that we can reduce the number of steps needed to execute the evolution rules that correspond to the `READ` and `HALT` instructions.

Recent universality results, obtained by simulating register machines, used spiking neural P systems. To our knowledge, the universality results for spiking neural P systems have not considered register machine models with `READ` instructions. Hence, we could consider implementing evolution rules of spiking neural P systems that simulate the `READ` instructions.

Acknowledgments

The authors wish to thank Radu Nicolescu for his inspiration in membrane computing and comments that helped us improve the paper. Parts of this paper were verified by software tools developed by Ivy Jiang (register machine simulator) and Aniruddh Gandhi and Habib Naderi (syntax translator).

References

[BG05] Francesco Bernardini and Marian Gheorghe. Cell communication in tissue P systems: universality results. *Soft Comput.*, 9(9):640–649, 2005.

[CC10a] Cristian S. Calude and Elena Calude. The complexity of the four colour theorem. *LMS J. Comput. Math.*, 13:414–425, 2010.

[CC10b] Cristian S. Calude and Elena Calude. Evaluating the complexity of mathematical problems. part 1. *Complex Systems*, 18:387–401, 2010.

[CCD06] Cristian S. Calude, Elena Calude, and Michael J. Dinneen. A new measure of the difficulty of problems. *Journal of Multiple-Valued Logic and Soft Computing*, 12:285–307, January 2006.

[CCM+11] Mónica Cardona, M. Angels Colomer, Antoni Margalida, Antoni Palau, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, and Delfí Sanuy. A computational modeling for real ecosystems based on P systems. *Natural Computing*, 10(1):39–53, 2011.

[CD07] Cristian S. Calude and Michael J. Dinneen. Exact approximations of Omega numbers. *Intl. J. of Bifurcation and Chaos*, 17(6):1937–1954, July 2007.

[CDK02] Gabriel Ciobanu, Rahul Desai, and Akash Kumar. Membrane systems and distributed computing. In Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *WMC-CdeA*, volume 2597 of *Lecture Notes in Computer Science*, pages 187–202. Springer-Verlag, 2002.

[Cha87] Gregory J. Chaitin. *Algorithmic Information Theory*. Cambridge University Press, Cambridge, UK, 1987.

[CPPPJ07] Gabriel Ciobanu, Linqiang Pan, Gheorghe Păun, and Mario J. Pérez-Jiménez. P systems with minimal parallelism. *Theor. Comput. Sci.*, 378(1):117–130, 2007.

[CVMVV07] Erzsébet Csuhaj-Varjú, Maurice Margenstern, György Vaszil, and Sergey Verlan. On small universal antiport P systems. *Theor. Comput. Sci.*, 372(2-3):152–164, 2007.

[Din12] Michael J. Dinneen. A program-size complexity measure for mathematical problems and conjectures. In Michael J. Dinneen, Bakhadyr Khoussainov, and André Nies, editors, *Computation, Physics and Beyond*, volume 7160 of *Lecture Notes in Computer Science*, pages 81–93. Springer, 2012. Presentation slides of WTCS2012 at `http://www.cs.auckland.ac.nz/research/conferences/wtcs2012/`.

[DKN10a] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. Edge- and vertex-disjoint paths in P modules. In Gabriel Ciobanu and Maciej Koutny, editors, *Workshop on Membrane Computing and Biologically Inspired Process Calculi*, pages 117–136, 2010.

[DKN10b] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. A faster P solution for the Byzantine agreement problem. In Marian Gheorghe, Thomas Hinze, and Gheorghe Păun, editors, *Conference on Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 175–197. Springer-Verlag, Berlin Heidelberg, 2010.

[DKN12] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. Faster synchronization in P systems. *Natural Computing*, 11:107–115, 2012.

[FKOS05] Rudolf Freund, Lila Kari, Marion Oswald, and Petr Sosík. Computationally universal P systems without priorities: two catalysts are sufficient. *Theor. Comput. Sci.*, 330(2):251–266, 2005.

[GID09] Marian Gheorghe, Florentin Ipate, and Ciprian Dragomir. Formal verification and testing based on P systems. In Păun et al. [PPJRN+10], pages 54–65.

[GILD10] Marian Gheorghe, Florentin Ipate, Raluca Lefticaru, and Ciprian Dragomir. An integrated approach to P systems formal verification. In Marian Gheorghe, Thomas Hinze, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Int. Conf. on Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 226–239. Springer, 2010.

[GPR+12] Marian Gheorghe, Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Sergey Verlan, editors. *Membrane Computing—12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers*, volume 7184 of *Lecture Notes in Computer Science*. Springer, 2012.

[GQGEPH+09] Manuel García-Quismondo, Rosa Gutiérrez-Escudero, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez. An overview of P-lingua 2.0. In Păun et al. [PPJRN+10], pages 264–288.

[II04] Tseren-Onolt Ishdorj and Mihai Ionescu. Replicative—distribution rules in P systems with active membranes. In Zhiming Liu and Keijiro Araki, editors, *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2004.

[ILPH+11] Florentin Ipate, Raluca Lefticaru, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, and Cristina Tudose. Formal verification of P systems with active membranes through model checking. In Gheorghe et al. [GPR+12], pages 215–225.

[ILPW09] Tseren-Onolt Ishdorj, Alberto Leporati, Linqiang Pan, and Jun Wang. Solving NP-complete problems by spiking neural P systems with budding rules. In Păun et al. [PPJRN+10], pages 335–353.

[IPY06] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural P systems. *Fundam. Inform.*, 71(2-3):279–308, 2006.

[MVPPRP03] Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems. *Theor. Comput. Sci.*, 296(2):295–326, 2003.

[Nic11] Radu Nicolescu. Parallel and distributed algorithms in P systems. In Gheorghe et al. [GPR+12], pages 35–50.

[Pău02] Gheorghe Păun. *Membrane Computing: An Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[Pău06] Gheorghe Păun. Introduction to membrane computing. In Gabriel Ciobanu, Mario J. Pérez-Jiménez, and Gheorghe Păun, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 1–42. Springer-Verlag, 2006.

[PP05] Gheorghe Păun and Radu A. Păun. Membrane computing as a framework for modeling economic processes. In *SYNASC*, pages 11–18. IEEE Computer Society, 2005.

[PP07] Andrei Păun and Gheorghe Păun. Small universal spiking neural P systems. *Biosystems*, 90(1):48–60, 2007.

[PPJRN+10] Gheorghe Păun, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, Grzegorz Rozenberg, and Arto Salomaa, editors. *Membrane Computing, 10th International Workshop, WMC 2009, Curtea de Argeş, Romania, August 24-27, 2009. Revised Selected and Invited Papers*, volume 5957 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[PRS10] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Inc., New York, NY, USA, 2010.

[ZFM00] Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri. Solving NP-complete problems using P systems with active membranes. In Ioannis Antoniou, Cristian S. Calude, and Michael J. Dinneen, editors, *UMC*, pages 289–301. Springer-Verlag, 2000.

A Appendix

This section contains the evolution rules of a simple P system, which corresponds to the register machine of Example 4 that computes the GCD of two integer numbers. The evolution rules of the provider cell are described in Section 3.2, hence, we do not list the provider cell's rules here. The evolution rules of the main cell are as follow. Note, we verified the correctness of the following evolution rules by our P system simulator.

Rules for initializing registers $a, b, c, d, e, f, g, h, i$ with the instruction line numbers (to be used as targets in branching `EQ` instructions)

- Rules for state s_0:

  1 $s_0\ a \rightarrow_{\texttt{min}} s_1\ a^{10}$

  2 $s_0\ a \rightarrow_{\texttt{max}} s_1$

- Rules for state s_1:

  1 $s_1\ b \rightarrow_{\texttt{min}} s_2\ b^{14}$

  2 $s_1\ b \rightarrow_{\texttt{max}} s_2$

- Rules for state s_2:

  1 $s_2\ c \rightarrow_{\texttt{min}} s_3\ c^{18}$

  2 $s_2\ c \rightarrow_{\texttt{max}} s_3$

- Rules for state s_3:

  1 $s_3\ d \rightarrow_{\texttt{min}} s_4\ d^{21}$

  2 $s_3\ d \rightarrow_{\texttt{max}} s_4$

- Rules for state s_4:

  1 $s_4\ e \rightarrow_{\texttt{min}} s_5\ e^{27}$

  2 $s_4\ e \rightarrow_{\texttt{max}} s_5$

- Rules for state s_5:

  1 $s_5\ f \rightarrow_{\texttt{min}} s_6\ f^{29}$

  2 $s_5\ f \rightarrow_{\texttt{max}} s_6$

- Rules for state s_6:

  1 $s_6\ g \rightarrow_{\texttt{min}} s_7\ g^{31}$

  2 $s_6\ g \rightarrow_{\texttt{max}} s_7$

- Rules for state s_7:

  1 $s_7\ h \rightarrow_{\texttt{min}} s_8\ h^{35}$

  2 $s_7\ h \rightarrow_{\texttt{max}} s_8$

- Rules for state s_8:

  1 $s_8\ i \rightarrow_{\texttt{min}} s_9\ i^{37}$

  2 $s_8\ i \rightarrow_{\texttt{max}} s_9$

Rules for initializing registers x and y, using auxiliary register z, with the first and the second values of the input data, respectively

- Rules for state s_9:

  1 $s_9\ z \rightarrow_{\texttt{min}} s'_9\ z\ (\mu, \downarrow)$

  2 $s_9\ z \rightarrow_{\texttt{max}} s'_9$

- Rules for state s'_9:

  1 $s'_9\ z \rightarrow_{\texttt{min}} s''_9\ z$

- Rules for state s''_9:

  1 $s''_9\ \phi \rightarrow_{\texttt{min}} s''_9\ \phi$

  2 $s''_9\ \delta \rightarrow_{\texttt{min}} s_{10}\ z$

  3 $s''_9\ z \rightarrow_{\texttt{min}} s_{10}\ z$

- Rules for state s_{10}:

  1 $s_{10}\ z^3 \rightarrow_{\texttt{min}} s_{11}\ z^3$

  2 $s_{10}\ z^2 \rightarrow_{\texttt{min}} s\ z^2$

  3 $s_{10}\ z \rightarrow_{\texttt{min}} s_{11}\ z$

  4 $s_{10}\ b \rightarrow_{\texttt{max}} s\ b\ \theta$

- Rules for state s_{11}:

  1 $s_{11}\ x \rightarrow_{\texttt{min}} s_{12}\ x^2$

- Rules for state s_{12}:

  1 $s_{12}\ a \rightarrow_{\texttt{max}} s\ a\ \theta$

- Rules for state s_{13}:

  1 $s_{13}\ z \rightarrow_{\texttt{min}} s'_{13}\ z\ (\mu, \downarrow)$

  2 $s_{13}\ z \rightarrow_{\texttt{max}} s'_{13}$

- Rules for state s'_{13}:

  1 $s'_{13}\ z \rightarrow_{\texttt{min}} s''_{13}\ z$

- Rules for state s''_{13}:

  1 $s''_{13}\ \phi \rightarrow_{\texttt{min}} s''_{13}\ \phi$

  2 $s''_{13}\ \delta \rightarrow_{\texttt{min}} s_{14}\ z$

  3 $s''_{13}\ z \rightarrow_{\texttt{min}} s_{14}\ z$

- Rules for state s_{14}:

  1 $s_{14}\ z^3 \rightarrow_{\texttt{min}} s_{15}\ z^3$

  2 $s_{14}\ z^2 \rightarrow_{\texttt{min}} s\ z^2$

  3 $s_{14}\ z \rightarrow_{\texttt{min}} s_{15}\ z$

  4 $s_{14}\ c \rightarrow_{\texttt{max}} s\ c\ \theta$

- Rules for state s_{15}:

  1 $s_{15}\ y \rightarrow_{\texttt{min}} s_{16}\ y^2$

- Rules for state s_{16}:

  1 $s_{16}\ b \rightarrow_{\texttt{max}} s\ b\ \theta$

Rules for checking the condition $x = 0$

- Rules for state s_{17}:

  1 $s_{17}\ x^2 \rightarrow_{\texttt{min}} s_{18}\ x^2$

  2 $s_{17}\ x^1 \rightarrow_{\texttt{min}} s\ x^1$

  3 $s_{17}\ x \rightarrow_{\texttt{min}} s_{18}\ x$

  4 $s_{17}\ i \rightarrow_{\texttt{max}} s\ i\ \theta$

Rules for checking the condition $x = 0$

- Rules for state s_{18}:

  1 $s_{18}\ z \rightarrow_{\texttt{max}} s_{19}$

  2 $s_{18}\ x \rightarrow_{\texttt{max}} s_{19}\ z\ x$

- Rules for state s_{19}:

  1 $s_{19}\ w \rightarrow_{\texttt{max}} s_{20}$

  2 $s_{19}\ y \rightarrow_{\texttt{max}} s_{20}\ w\ y$

- Rules for state s_{20}:

  1 $s_{20}\ z\ y \rightarrow_{\texttt{max}} s'_{20}\ z\ y$

  2 $s_{20}\ z \rightarrow_{\texttt{max}} s'_{20}\ z\ \gamma$

  3 $s_{20}\ y \rightarrow_{\texttt{max}} s'_{20}\ z\ \gamma$

- Rules for state s'_{20}:

  1 $s'_{20}\ \gamma \rightarrow_{\texttt{max}} s_{21}$

  2 $s'_{20}\ f \rightarrow_{\texttt{max}} s\ f\ \theta$

- Rules for state s_{21}:

  1 $s_{21}\ z \rightarrow_{\texttt{min}} s_{22}\ z^2$

- Rules for state s_{22}:

  1 $s_{22}\ w \rightarrow_{\texttt{min}} s_{23}\ w^2$

- Rules for state s_{23}:

  1 $s_{23}\ y\ z \rightarrow_{\texttt{max}} s'_{23}\ y\ z$

  2 $s_{23}\ y \rightarrow_{\texttt{max}} s'_{23}\ y\ \gamma$

  3 $s_{23}\ z \rightarrow_{\texttt{max}} s'_{23}\ y\ \gamma$

- Rules for state s'_{23}:

  1 $s'_{23}\ \gamma \rightarrow_{\texttt{max}} s_{24}$

  2 $s'_{23}\ e \rightarrow_{\texttt{max}} s\ e\ \theta$

- Rules for state s_{24}:

  1 $s_{24}\ x\ w \rightarrow_{\texttt{max}} s'_{24}\ x\ w$

  2 $s_{24}\ x \rightarrow_{\texttt{max}} s'_{24}\ x\ \gamma$

  3 $s_{24}\ w \rightarrow_{\texttt{max}} s'_{24}\ x\ \gamma$

- Rules for state s'_{24}:

  1 $s'_{24}\ \gamma \rightarrow_{\texttt{max}} s_{25}$

  2 $s'_{24}\ f \rightarrow_{\texttt{max}} s\ f\ \theta$

- Rules for state s_{25}:

  1 $s_{25}\ d \rightarrow_{\texttt{max}} s\ d\ \theta$

- Rules for state s_{26}:

  1 $s_{26}\ y \rightarrow_{\texttt{max}} s_{27}$

  2 $s_{26}\ x \rightarrow_{\texttt{max}} s_{27}\ y\ x$

- Rules for state s_{27}:

  1 $s_{27}\ x \rightarrow_{\texttt{max}} s_{28}$

  2 $s_{27}\ z \rightarrow_{\texttt{max}} s_{28}\ x\ z$

Rules for executing $x := x - y$

- Rules for state s_{28}:

  1 $s_{28}\ z \rightarrow_{\texttt{max}} s_{29}$

  2 $s_{28}\ y \rightarrow_{\texttt{max}} s_{29}\ z\ y$

- Rules for state s_{29}:

  1 $s_{29}\ w \rightarrow_{\texttt{min}} s_{30}\ w^{1}$

  2 $s_{29}\ w \rightarrow_{\texttt{max}} s_{30}$

- Rules for state s_{30}:

  1 $s_{30}\ x\ z \rightarrow_{\texttt{max}} s'_{30}\ x\ z$

  2 $s_{30}\ x \rightarrow_{\texttt{max}} s'_{30}\ x\ \gamma$

  3 $s_{30}\ z \rightarrow_{\texttt{max}} s'_{30}\ x\ \gamma$

- Rules for state s'_{30}:

  1 $s'_{30}\ \gamma \rightarrow_{\texttt{max}} s_{31}$

  2 $s'_{30}\ h \rightarrow_{\texttt{max}} s\ h\ \theta$

- Rules for state s_{31}:

  1 $s_{31}\ z \rightarrow_{\texttt{min}} s_{32}\ z^{2}$

- Rules for state s_{32}:

  1 $s_{32}\ w \rightarrow_{\texttt{min}} s_{33}\ w^{2}$

- Rules for state s_{33}:

  1 $s_{33}\ g \rightarrow_{\texttt{max}} s\ g\ \theta$

- Rules for state s_{34}:

  1 $s_{34}\ x \rightarrow_{\texttt{max}} s_{35}$

  2 $s_{34}\ w \rightarrow_{\texttt{max}} s_{35}\ x\ w$

- Rules for state s_{35}:

  1 $s_{35}\ c \rightarrow_{\texttt{max}} s\ c\ \theta$

Rules for executing Halt

- Rules for state s_{36}:

  1 $s_{36}\ \pi \rightarrow_{\texttt{min}} s'_{36}\ \pi\ (\mu,\downarrow)\ (\mu,\downarrow)$

- Rules for state s'_{36}:

  1 $s'_{36}\ \pi \rightarrow_{\texttt{min}} s''_{36}\ \pi$

- Rules for state s''_{36}:

  1 $s''_{36}\ \phi\ \pi \rightarrow_{\texttt{min}} s_{36}$

  2 $s''_{36}\ \pi \rightarrow_{\texttt{min}} s''_{36}\ \pi$

Rules for the "springboard" state s

- Rules for state s:

  1 $s\ \theta^{37} \rightarrow_{\texttt{min}} s\ \theta^{37}$

  2 $s\ \theta^{36} \rightarrow_{\texttt{min}} s_{35}$

  3 $s\ \theta^{35} \rightarrow_{\texttt{min}} s_{34}$

  $\vdots$

  37 $s\ \theta \rightarrow_{\texttt{min}} s_{0}$