CDMTCS Research Report Series

Simulation of Functional Register Machines using Active P Systems

Michael J. Dinneen
Yun-Bum Kim

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

CDMTCS-466
July 2014

Centre for Discrete Mathematics and
Theoretical Computer Science

Simulation of Functional Register Machines using Active P Systems

Michael J. Dinneen and Yun-Bum Kim

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

mjd@cs.auckland.ac.nz, tkim021@aucklanduni.ac.nz

Abstract

Active P systems are a bio-inspired distributed and parallel computation model, consisting of network of computing units called membranes, where membranes can be added and removed during the computation. This paper presents the simulation of functional register machines (i.e. a register machine model that includes instructions that can define functions and make function calls) using active P systems with the same run-time complexity.

Keywords: P systems, register machines, functional programming, universal computer

1 Introduction

Membrane systems [9, 11] (also known as P systems) are distributed and parallel computing model, inspired by the structure and function of living cells. A membrane system consists of a network of (multiset processing) computing units called membranes. Each membrane contains a multiset of symbols and is associated with a set of multiset processing rules. Several variant P system models [8, 7] have been introduced, inspired from various features of living cells, that provide new ways to process information and solve the computational problems of interest. An *active P system* [10] is a variant P system model that supports dynamic network structure of membranes by: (i) adding new membranes to the systems and (ii) removing existing membranes from the system. An active P system model [10] extends a transition P system model [10] by incorporating *membrane creation* operation (which adds new membranes to the system) and *membrane dissolution* operation (which removes existing membranes from the system). Figure 1 illustrates creation and dissolution of membranes; a child membrane, μ_j, can be created inside membrane μ_i with content w_j, and membrane μ_j can dissolve, leaving its content, w_j, to its parent membrane, μ_i. These operations are incorporated into evolution rule specification, such that executing evolution rules with these operations can create or dissolve membranes of a system.

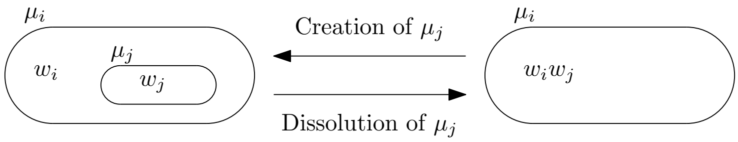

Figure 1: Membrane creation and dissolution operations [10].

In theoretical computer science, register machines are computational models with the equivalent computation power of a Turing machine. Register machines [2] have a finite set of instructions that can perform data handling, arithmetic, logic and control flow operations. A recently introduced functional register machine [1] extends the model of [2] by including instructions for defining functions and making function calls.

In the P systems community, several studies have shown *Turing completeness* (i.e. can compute anything a Turing machine [12] can compute) and *universality* (i.e. can simulate an arbitrary Turing machine with input) results of various P system models by showing that these P system models can simluate register machines [6, 7, 4, 5].

The main results of this paper is the simulation of functional register machines using active P systems with the same run-time complexity. For the recently introduced functional register machine model [1], which is more complex than the ones considered in the earlier studies, we present the details for constructing an active P system that simulates an arbitrary functional register machine. Specifically, we present a set of evolution rules that replicate the behavior of each of the instructions of this functional register machine with the same run-time complexity.

This paper is organized as follows. Section 2 recalls the definitions of a functional register machine model and a P system with active membrane model, and covers several key mathematical concepts. Section 3 presents the details of constructing an active P system that simulates any functional register machine. Finally, Section 4 summarizes this paper.

2 Preliminaries

2.1 Strings, multisets and graphs

An *alphabet* is a finite non-empty set with elements called *symbols*. A *string over* alphabet O is a finite sequence of symbols from O. The set of all strings over O is denoted by O^*. The length of a string $x \in O^*$, denoted by $|x|$, is the number of symbols in x. The number of occurrences of a symbol $o \in O$ in a string x over O is denoted by $|x|_o$. The *empty string* is denoted by λ.

A *multiset* is a set with multiplicities associated with its elements. A set that contains the distinct elements of a multiset v is denoted by $\texttt{distinct}(v)$. The empty string or multiset is represented by λ. The *size* of a multiset v is denoted by $|v|$. The *multiplicity* of an element x in a multiset v is denoted by $|v|_x$. We say that a multiset v is *included* in a multiset w, denoted by $w \subseteq v$, if, for all $o \in O$, $|w|_o \leq |v|_o$. The *union* of multisets v and w, denoted by $v \cup w$, is a multiset x, such that, for all $o \in O$, $|x|_o = |v|_o + |w|_o$. The *difference* of multisets v and w, denoted by $v - w$, is a multiset x, such that, for all

$o \in O$, $|x|_o = \max(|v|_o - |w|_o, 0)$.

A (binary) *relation* R over two sets X and Y is a subset of their Cartesian product, $R \subseteq X \times Y$. For $A \subseteq X$ and $B \subseteq Y$, we set $R(A) = \{y \in Y \mid \exists x \in A, (x, y) \in R\}$, $R^{-1}(B) = \{x \in X \mid \exists y \in B, (x, y) \in R\}$.

A *graph* is an ordered pair (V, E), where V is a finite set of elements called nodes and E is a set of unordered pairs of V called edges. A *path* of length $n-1$ is a sequence of n nodes, $v_1, v_2, \ldots, v_n$, such that $\{(v_1, v_2), \ldots, (v_{n-1}, v_n)\} \subseteq E$. The *diameter* of G, denoted by $\texttt{dia}(G)$, is the maximum of the lengths of shortest paths between every pair of nodes of G.

A *directed graph* (digraph) is a pair (V, A), where V is a finite set of elements called nodes and A is a set of an ordered pair of V called *arcs*. Given a digraph $D = (V, A)$, for $v \in V$, the *parents* of v are $A^{-1}(v) = A^{-1}(\{v\})$ and the *children* of v are $A(v) = A(\{v\})$.

2.2 Register machines

A register machine has $n \geq 1$ instructions and $m \geq 0$ registers, where each register may hold an arbitrarily large non-negative integer. All registers are, by default, initialized to 0. A register machine program consists of a finite list of instructions, followed by optional *input data*, denoted as a sequence of bits. The first instruction of a program is indexed at address (i.e. line number) 0.

A set of instructions of a register machine [1], denoted in Chaitin's style [3], is described below, which perform data handling, arithmetic, logic and control flow operations. In the instructions below, variables z_1, z_2 and z_3 denote registers and k denotes a non-negative binary integer constant. The content of register z_i, $1 \leq i \leq 3$, is denoted by $\texttt{value}(z_i)$.

- Instruction ($\texttt{EQ}$ r_1 r_2 r_3) or ($\texttt{EQ}$ r_1 k r_3):
  If $\texttt{value}(r_1) = \texttt{value}(r_2)$ or $\texttt{value}(r_1) = k$, then the execution of M continues at the $\texttt{value}(r_3)$-th next instruction in the sequence. Otherwise, the execution of M continues at the next instruction.

- Instruction ($\texttt{EQ}$ r_1 r_2 $-r_3$) or ($\texttt{EQ}$ r_1 k $-r_3$):
  If $\texttt{value}(r_1) = \texttt{value}(r_2)$ or $\texttt{value}(r_1) = k$, then the execution of M continues at the $\texttt{value}(r_3)$-th previous instruction in the sequence. Otherwise, the execution of M continues at the next instruction.

- Instruction ($\texttt{SET}$ r_1 r_2) or ($\texttt{SET}$ r_1 k):
  r_1 is replaced by $\texttt{value}(r_2)$ or the constant k.

- Instruction ($\texttt{ADD}$ r_1 r_2) or ($\texttt{ADD}$ r_1 k):
  r_1 is replaced by $\texttt{value}(r_1) + \texttt{value}(r_2)$ or $\texttt{value}(r_1) + k$.

- Instruction ($\texttt{READ}$ r_1):
  r_1 is replaced by x, where x is the integer value of the binary input data with leading bit of 1.

- Instruction (`HALT`):
  This is the last instruction of a register machine program, which is used to separate program instructions from binary input data.

Functional register machine model of [1] includes the following additional instructions, which can define functions and make function calls.

- Instruction (`FUNC` f r_1):
  Declares a function, named f, that takes r_1 as an input argument. It is assumed that the `FUNC` instructions will only be executed via previous `CALL` instructions.

- Instruction (`CALL` f r_1 r_2):
  Makes a function call to the function f, which: (i) passes r_1 as an input argument of f, (ii) sets the content of r_2 with the return value of f and (iii) on returning, restores every register, except register r_2, to its original content prior to this function call.

- Instruction (`RETURN` r_1):
  This instruction corresponds to one of "return" statements of a function, which returns a value (i.e. the content of r_1) back to where a function call was made. The execution of the register machine continues to the instruction that made a function call. The main program may also halt using this instruction.

2.2.1 Register machine subroutine

The *Cantor pairing function*, $\texttt{cantor} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, is defined by $\texttt{cantor}(x, y) = 1/2(x + y)(x + y + 1) + y$. The inverse this `cantor` is denoted by $\texttt{cantor}^{-1}$.

In the register machine model [1], the Cantor pairing function defined below is used to store and retrieve a list of integers. The Cantor pairing function is denoted by (`CNTR` r_1 r_2 r_3), where r_1, r_2 and r_3 denote registers.

- If $\texttt{value}(r_3) = 0$, set $\texttt{value}(r_3)$ as $\texttt{cantor}(\texttt{value}(r_1), \texttt{value}(r_2))$.
- Otherwise, compute $(\texttt{value}(r_1), \texttt{value}(r_z)) = \texttt{cantor}^{-1}(\texttt{value}(r_3))$.

An implementation of the (`CNTR` r_1 r_2 r_3) instruction is given below, which uses the basic instructions and registers a, t_1, t_2, t_3.

Line	Instruction	Line	Instruction	Line	Instruction
0	`EQ` r_3 0 9	9	`ADD` t_1 t_2	18	`EQ` a a -9
1	`SET` t_2 r_1	10	`SET` t_4 t_1	19	`SET` r_2 t_3
2	`ADD` t_2 r_2	11	`SET` t_3 0	20	`SET` r_1 0
3	`ADD` r_3 t_1	12	`EQ` r_3 t_4 7	21	`EQ` t_2 t_3 4
4	`EQ` t_1 t_2 3	13	`EQ` t_3 t_2 4	22	`ADD` r_1 1
5	`ADD` t_1 1	14	`ADD` t_3 1	23	`ADD` t_3 1
6	`EQ` a a -3	15	`ADD` t_4 1	24	`EQ` a a -3
7	`ADD` r_3 r_2	16	`EQ` a a -4		
8	`EQ` a a 17	17	`ADD` t_2 1		

2.3 Active P systems

An active P system of order n is $\Pi = (O, K, \Delta)$, where

1. O is a finite non-empty alphabet of *symbols*.
2. $K = \{\mu_1, \mu_2, \ldots, \mu_n\}$ is a finite set of *membranes*. Each $\mu_i \in K$ is of the form $\mu_i = (Q_i, s_{i0}, w_{i0}, R_i)$, 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. An evolution rule $r \in R_i$ has the form:

     $$j \ s \ u \rightarrow_\alpha s' \ v \ w \ x$$

     where
     - $\alpha \in \{\mathtt{min}, \mathtt{max}\}$ 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 *start state* and s' is the *target state* of r,
     - $u \in O^+$,
     - $v \in (O \times \tau)^*$, where $\tau \in \{\odot, \uparrow, \downarrow, \updownarrow\}$ is a *target indicator*. Note that, $(o, \odot) \in v$, $o \in O$, is abbreviated to o,
     - $w \in ([s, x])^*$, where $[s, x]$ is the notation used to create a child membrane inside μ_i with an initial state $s \in Q_i$ and an initial content $x \in O^*$,
     - $x \in \{\lambda, \delta\}$, where δ is the notation used to remove the current membrane μ_i from system Π.
3. Δ is an irreflexive and asymmetric relation on K, representing a set of arcs between membranes with bidirectional communication capabilities.

In each step, each membrane $\mu_i = (Q_i, s_i, w_i, R_i)$ finds a multiset of rules, M_i, as follows. Let $U_i = \bigcup_{r_h \in M_i} \mathtt{LHS}(r_h)$. For each rule $r_j \in R_i$, $1 \leq j \leq |R_i|$ (in an increasing priority order), membrane μ_i adds $k \geq 1$ copies of rule r_j into M_i, if:

- $\mathtt{source}(r_j) = s_i$,
- $\mathtt{LHS}(r_j) \subseteq w_i \setminus U_i$ and
- $\mathtt{target}(r_j)$ equals the target states of all the rules in M_i, where:
  - $k = 1$, if $\alpha = \mathtt{min}$ and
  - $k = t$, such that $\mathtt{LHS}(r_j)^{t+1} \nsubseteq w_i \setminus U_i$, if $\alpha = \mathtt{max}$.

Each membrane $\mu_i = (Q_i, s_i, w_i, R_i)$ applies all the rules of M_i simultaneously and performs the following:

- transits from state s_i to $\mathtt{target}(r_i)$, $r_i \in M_i$,

- transforms multiset w_i into multiset $w'_i = \bigcup\{o \mid (o, \odot) \in \texttt{RHS}(r_j), r_j \in M_i\} \cup V_i$, where V_i corresponds to the multiset of symbols that membrane μ_i receives from its neighbors, i.e. $\Delta(i) \cup \Delta^{-1}(i)$,
- sends one copy of symbol o of $(o, \tau) \in \texttt{RHS}(r_j)$, $r_j \in M_i$, to:
    - each membrane $\mu_j \in \Delta^{-1}(i)$, if $\tau = \uparrow$,
    - each membrane $\mu_j \in \Delta(i)$, if $\tau = \downarrow$ and
    - each membrane $\mu_j \in \Delta(i) \cup \Delta^{-1}(i)$, if $\tau = \updownarrow$,
- creates a child membrane if M_i contains a rule with notation "[ ]",
- dissolves μ_i if there is a rule in M_i that contains symbol δ.

A system *halts*, if no further rules are applicable for all membranes. The *computational results* of a halted system are the multiplicities of symbols present in the membranes of the system.

3 Register machine simulator

We assume that, in a register machine program compiled using the register machine model of Section 2.2, the `FUNC` instructions will only be executed via previous `CALL` instructions.

Given an arbitrary register machine M of Section 2.2, with $n \geq 1$ instructions and $k \geq 0$ registers, $r_0, r_1, \ldots, r_{k-1}$, we build a P system $\Pi_M = (O, K, \Delta)$ that simulates M, where:

1. $O = \{r_i \mid 0 \leq i < k\} \cup \{*, +, \phi\}$
    - the multiplicity of symbol r_i minus one, $0 \leq i < k$, represents the value of register r_i,
    - the multiplicity of symbol $*$ minus one equals the current instruction line index, i.e. multiset $*^{j+1}$, $0 \leq j < n$, represents j-th instruction.
    - $+$ is an auxiliary symbols used for executing the evolution rules that correspond to the `EQ`, `CALL`, `RETURN` and `FUNC` instructions.
    - the multiplicity of symbol ϕ minus one equals the integer value of the binary input data with leading bit of 1.
2. $K = \{\mu_m\}$, where membrane μ_m is of the form

$$(Q_m, s_{m0}, w_{m0}, R_m)$$

    - $Q_m = \{s_i, s'_i \mid 0 \leq i < n\} \cup \{s_{\texttt{GOTO}}\} \cup \{s_f \mid$ for every function (`FUNC` f z_{i_1}) included in a given register machine program$\}$, where:
        - s_i, s'_i, $0 \leq i < n$, represent the i-th instruction of M,
        - s_{n-1} represent the "halting" state,

- ◦ $s_{\texttt{GOTO}}$ represents the "GOTO" state; if the execution of M continues to j-th instruction, $0 \leq j < n$, then the "GOTO" state enables μ_m to transit to state s_j.

- $s_{m0} = s_0$, indicates the first instruction, i.e. 0-th instruction.
- $w_{m0} = \{r_i \mid 0 \leq i < k\} \cup \{*\} \cup \{\phi^{z+1} \mid z$ is the integer value of the binary input data with leading bit of 1$\}$, indicates μ_m's initial content.
- R_m corresponds to a set of evolution rules that replicate the behavior of the instructions of M. The rules of R_m are described in the following subsections.

3. $\Delta = \emptyset$.

3.1 Evolution rules for a SET instruction

3.1.1 Rules for an i-th instruction of the form (SET r_{i_1} r_{i_2})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_2 \geq 1$ copies of symbol r_{i_2} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:**
  1. $s_i \; * \rightarrow_{\texttt{min}} s_{i+1} \; * \; *$
  2. $s_i \; r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}$
  3. $s_i \; r_{i_2} \rightarrow_{\texttt{max}} s_{i+1} \; r_{i_1} \; r_{i_2}$
- **Postcondition:** End state is s_{i+1}. n_2 copies of symbol r_{i_1}, n_2 copies of symbol r_{i_1} and $n_4 + 1$ copies of symbol $*$.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 consumes all copies of symbol r_{i_1}. At the same time, rule 3 rewrites every copy of symbol r_{i_2} into multiset $r_{i_1} r_{i_2}$.

3.1.2 Rules for an i-th instruction of the form (SET r_{i_1} k_i)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:**
  1. $s_i \; * \rightarrow_{\texttt{min}} s_{i+1} \; * \; *$
  2. $s_i \; r_{i_1} \rightarrow_{\texttt{min}} s_{i+1} \; r_{i_1}^{k_i+1}$
  3. $s_i \; r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}$
- **Postcondition:** End state is s_{i+1}. $k_i + 1$ copies of symbol r_{i_1} and $n_4 + 1$ copies of symbol $*$.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 rewrites one copy of symbol r_{i_1} into $k_i + 1$ copies of symbol r_{i_1}. Rule 3 consumes the remaining copies of symbol r_{i_1}.

3.2 Evolution rules for an ADD instruction

3.2.1 Rules for an i-th instruction of the form (ADD r_{i_1} r_{i_2})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_2 \geq 1$ copies of symbol r_{i_2} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:**

  1. $s_i \; * \rightarrow_{\texttt{min}} s_{i+1} \; * \; *$
  2. $s_i \; r_{i_2} \rightarrow_{\texttt{min}} s_{i+1} \; r_{i_2}$
  3. $s_i \; r_{i_2} \rightarrow_{\texttt{max}} s_{i+1} \; r_{i_1} \; r_{i_2}$

- **Postcondition:** End state is s_{i+1}. $n_1 + n_2 - 1$ copies of symbol r_{i_1}, n_2 copies of symbol r_{i_2} and $n_4 + 1$ copies of symbol $*$.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 rewrites one copy of symbol r_{i_2} into one copy of symbol r_{i_2}. Rule 3 rewrites every remaining copy of symbol r_{i_2} into multiset $r_{i_1} r_{i_2}$.

3.2.2 Rules for an i-th instruction of the form (ADD r_{i_1} k_i)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:**

  1. $s_i \; * \rightarrow_{\texttt{min}} s_{i+1} \; * \; *$
  2. $s_i \; r_{i_1} \rightarrow_{\texttt{min}} s_{i+1} \; r_{i_1}^{k_i+1}$

- **Postcondition:** End state is s_{i+1}. $n_1 + k_i$ copies of symbol r_{i_1} and $n_4 + 1$ copies of symbol $*$.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 rewrites one copy of symbol r_{i_1} into $k_i + 1$ copies of symbol r_{i_1}.

3.3 Evolution rules for an EQ instruction

This section presents the evolution rules that correspond to an i-th instruction of the form (EQ r_{i_1} r_{i_1} r_{i_3}), (EQ r_{i_1} r_{i_2} r_{i_3}), (EQ r_{i_1} k_i r_{i_3}), (EQ r_{i_1} r_{i_1} $-r_{i_3}$), (EQ r_{i_1} r_{i_2} $-r_{i_3}$) or (EQ r_{i_1} k_i $-r_{i_3}$). The rules of these instructions use the rules of the "GOTO" state (described below), which mimic the way a register machine makes a jump to a particular instruction.

3.3.1 Rules for the "GOTO" state

Recall that system Π_M keeps track of the current instruction line index by the multiplicity of symbol $*$, where multiset $*^{j+1}$ indicates instruction line j. In the "GOTO" state, membrane's state transition is determined by the multiplicity of symbol $*$, such that if a membrane contains $j+1$ copies of symbol $*$, then the membrane transits to state s_j. The rules of the "GOTO" state are given below.

1	$s_{\texttt{GOTO}} *^{n+1} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}} *^{n+1}$
2	$s_{\texttt{GOTO}} *^{n} \rightarrow_{\texttt{min}} s_{n-1} *^{n}$
3	$s_{\texttt{GOTO}} *^{n-1} \rightarrow_{\texttt{min}} s_{n-2} *^{n-1}$
$\vdots$	
n	$s_{\texttt{GOTO}} *^{2} \rightarrow_{\texttt{min}} s_{1} *^{2}$
$n+1$	$s_{\texttt{GOTO}} * \rightarrow_{\texttt{min}} s_{0} *$

3.3.2 Rules for an i-th instruction of the form (EQ r_{i_1} r_{i_1} r_{i_3})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_3 \geq 1$ copies of symbol r_{i_3} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:** Rules below plus the rules of the "GOTO" state

  1. $s_i\ r_{i_3} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}}\ r_{i_3}$
  2. $s_i\ r_{i_3} \rightarrow_{\texttt{max}} s_{\texttt{GOTO}}\ r_{i_3}\ *$

- **Postcondition:** n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and j copies of symbol $*$, where $j = n_3 + n_4 - 1$. End state is s_{j-1}.
- **Description:** Rule 1 rewrites one copy of symbol r_{i_3} into one copy of symbol r_{i_3}. For the remaining $n_3 - 1$ copies of symbol r_{i_3}, rule 2 rewrites every copy of symbol r_{i_3} into one copy of symbol r_{i_3} and one copy of symbol $*$. Then, using the $j = n_3 + n_4 - 1$ copies of symbol $*$, the "GOTO" state rules guide the cell to transit to state s_{j-1}.

3.3.3 Rules for an i-th instruction of the form (EQ r_{i_1} r_{i_2} r_{i_3})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_2 \geq 1$ copies of symbol r_{i_2}, $n_3 \geq 1$ copies of symbol r_{i_3}, $n_4 \geq 1$ copies of symbol $*$ and $n_5 \geq 0$ copies of symbol $+$.
- **Rules:** Rules below plus the rules of the "GOTO" state

  Rules of state s_i:

  1. $s_i\ + \rightarrow_{\texttt{max}} s'_i$
  2. $s_i\ r_{i_1}\ r_{i_2} \rightarrow_{\texttt{max}} s'_i\ r_{i_1}\ r_{i_2}$
  3. $s_i\ r_{i_1} \rightarrow_{\texttt{min}} s'_i\ r_{i_1}\ +$
  4. $s_i\ r_{i_2} \rightarrow_{\texttt{min}} s'_i\ r_{i_2}\ +$

  Rules of state s'_i:

  5. $s'_i\ + \rightarrow_{\texttt{min}} s_{i+1}\ *$
  6. $s'_i\ r_{i_3} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}}\ r_{i_3}$
  7. $s'_i\ r_{i_3} \rightarrow_{\texttt{max}} s_{\texttt{GOTO}}\ r_{i_3}\ *$

- **Postcondition:**
  - If the contents of registers r_{i_1} and r_{i_2} are the same, then:
    End state is s_{j-1}, where $j = n_3 + n_4 - 1$. n_1 copies of symbol r_{i_1}, n_2 copies of symbol r_{i_2}, n_3 copies of symbol r_{i_3}, j copies of symbol $*$ and zero copies of symbol $+$.

- Otherwise:
  End state is s_{i+1}. n_1 copies of symbol r_{i_1}, n_2 copies of symbol r_{i_2}, n_3 copies of symbol r_{i_3}, n_4+1 copies of symbol $*$ and zero copies of symbol $+$.

- **Description:** Rule 1 consumes all copies of symbol $+$, if any, such that one copy of symbol $+$ produced by rule 3 or 4 can indicate that the values of the registers r_{i_1} and r_{i_2} are not the same. Rule 2 pairs up every copy of symbol r_{i_1} with one copy of r_{i_2}. If there are any unpaired copies of symbol r_{i_1} or r_{i_2}, then rule 3 or 4 produces one copy of symbol $+$. If symbol $+$ is present (i.e. the contents of registers r_{i_1} and r_{i_2} are not the same), then rule 5 consumes symbol $+$ and sets target state to s_{i+1}. If symbol $+$ is not present (i.e. the contents of registers r_{i_1} and r_{i_2} are the same), then rules 6 and 7 produce j copies of symbol $*$, where $j = n_3 + n_4 - 1$, which are used by the "GOTO" state rules to guide the cell to transit to state to s_{j-1}.

3.3.4 Rules for an i-th instruction of the form ($\texttt{EQ}\ r_{i_1}\ k_i\ r_{i_3}$)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_3 \geq 1$ copies of symbol r_{i_3} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:** Rules below plus the rules of the "GOTO" state
  1. $s_i\ r_{i_1}^{k_i+2} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}^{k_i+2}$
  2. $s_i\ r_{i_1}^{k_i+1}\ * \rightarrow_{\texttt{min}} s_{\texttt{GOTO}}\ r_{i_1}^{k_i+1}$
  3. $s_i\ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1}\ r_{i_1}$
  4. $s_i\ r_{i_3} \rightarrow_{\texttt{max}} s_{\texttt{GOTO}}\ r_{i_3}\ *$
  5. $s_i\ * \rightarrow_{\texttt{min}} s_{i+1}\ *\ *$
- **Postcondition:**
  - If the contents of registers r_{i_1} equals the constant k_i, then:
    End state is s_{j-1}, where $j = n_3 + n_4 - 1$. n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and j copies of symbol $*$.
  - Otherwise:
    End state is s_{i+1}. n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and n_4+1 copies of symbol $*$.

- **Description:** Rules 1, 2 and 3 check conditions $r_{i_1} > k_i$, $r_{i_1} = k_i$ and $r_{i_1} < k_i$, respectively. If the condition of rule 2 is met, then rule 2, together with rule 4, produce j copies of symbol $*$, where $j = n_3 + n_4 - 1$, which are used by the "GOTO" state rules to guide the cell transits to state s_{j-1}. If the condition of rule 2 is not met, then rule 1 or rule 3 prompts the cell to transit to state s_{i+1} and rule 5 produces one additional copy of symbol $*$.

3.3.5 Rules for an i-th instruction of the form ($\texttt{EQ}\ r_{i_1}\ r_{i_1}\ -r_{i_3}$)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_3 \geq 1$ copies of symbol r_{i_3} and $n_4 \geq 1$ copies of symbol $*$.

- **Rules:** Rules below plus the rules of the "GOTO" state

    1. $s_i \ r_{i_3} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}} \ r_{i_3}$
    2. $s_i \ r_{i_3} \ * \rightarrow_{\texttt{max}} s_{\texttt{GOTO}} \ r_{i_3}$

- **Postcondition:** n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and j copies of symbol $*$, where $j = n_4 - n_3 + 1$. End state is s_{j-1}.
- **Description:** Rule 1 rewrites one copy of symbol r_{i_3} into one copy of symbol r_{i_3}. For the remaining $n_3 - 1$ copies of symbol r_{i_3}. Rule 2 rewrites one copy of symbol $*$ and copy of symbol r_{i_3} into one copy of symbol r_{i_3}. Then, using the $j = n_4 - n_3 + 1$ copies of symbol $*$ produced from rules 1 and 2, the rules of the "GOTO" state guide the cell to transit to state s_{j-1}.

3.3.6 Rules for an i-th instruction of the form ($\texttt{EQ} \ r_{i_1} \ r_{i_2} \ -r_{i_3}$)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_2 \geq 1$ copies of symbol r_{i_2}, $n_3 \geq 1$ copies of symbol r_{i_3}, $n_4 \geq 1$ copies of symbol $*$ and $n_5 \geq 0$ copies of symbol $+$.
- **Rules:** Rules below plus the rules of the "GOTO" state

    Rules of state s_i:

    1. $s_i \ + \rightarrow_{\texttt{max}} s'_i$
    2. $s_i \ r_{i_1} \ r_{i_2} \rightarrow_{\texttt{max}} s'_i \ r_{i_1} \ r_{i_2}$
    3. $s_i \ r_{i_1} \rightarrow_{\texttt{min}} s'_i \ r_{i_1} \ +$
    4. $s_i \ r_{i_2} \rightarrow_{\texttt{min}} s'_i \ r_{i_2} \ +$

    Rules of state s'_i:

    5. $s'_i \ + \rightarrow_{\texttt{min}} s_{i+1} \ *$
    6. $s'_i \ r_{i_3} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}} \ r_{i_3}$
    7. $s'_i \ r_{i_3} \ * \rightarrow_{\texttt{max}} s_{\texttt{GOTO}} \ r_{i_3}$

- **Postcondition:**
    - If the contents of registers r_{i_1} and r_{i_2} are the same, then:
      End state is s_{j-1}, where $j = n_4 - n_3 + 1$. n_1 copies of symbol r_{i_1}, n_2 copies of symbol r_{i_2}, n_3 copies of symbol r_{i_3}, j copies of symbol $*$ and zero copies of symbol $+$.
    - Otherwise:
      End state is s_{i+1}. n_1 copies of symbol r_{i_1}, n_2 copies of symbol r_{i_2}, n_3 copies of symbol r_{i_3}, $n_4 + 1$ copies of symbol $*$ and zero copies of symbol $+$.

- **Description:** Rule 1 consumes all copies of symbol $+$, if any, such that one copy of symbol $+$ produced by rule 3 or 4 can indicate that the values of the registers r_{i_1} and r_{i_2} are not the same. Rule 2 pairs up every copy of symbol r_{i_1} with one copy of r_{i_2}. If there are any unpaired copies of symbol r_{i_1} or r_{i_2}, then rule 3 or 4 produces one copy of symbol $+$. If symbol $+$ is present (i.e. the contents of registers r_{i_1} and r_{i_2} are not the same), then rule 5 consumes symbol $+$ and sets target state to s_{i+1}. If symbol $+$ is not present (i.e. the contents of registers r_{i_1} and r_{i_2} are the same), then rules 6 and 7 produce j copies of symbol $*$, where $j = n_4 - n_3 + 1$, which are used by the "GOTO" state rules to guide the cell to transit to state to s_{j-1}.

3.3.7 Rules for an i-th instruction of the form (EQ r_{i_1} k_i $-r_{i_3}$)

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_3 \geq 1$ copies of symbol r_{i_3} and $n_4 \geq 1$ copies of symbol $*$.
- **Rules:** Rules below plus the rules of the "GOTO" state

  1. $s_i \ r_{i_1}^{k_i+2} \rightarrow_{\texttt{min}} s_{i+1} \ r_{i_1}^{k_i+2}$
  2. $s_i \ r_{i_1}^{k_i+1} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}} \ r_{i_1}^{k_i+1}$
  3. $s_i \ r_{i_1} \rightarrow_{\texttt{min}} s_{i+1} \ r_{i_1}$
  4. $s_i \ r_{i_3} \rightarrow_{\texttt{min}} s_{\texttt{GOTO}} \ r_{i_3}$
  5. $s_i \ r_{i_3} \ * \rightarrow_{\texttt{max}} s_{\texttt{GOTO}} \ r_{i_3}$
  6. $s_i \ * \rightarrow_{\texttt{min}} s_{i+1} \ * \ *$

- **Postcondition:**

  - If the content of register r_{i_1} equals the constant k_i, then:
    End state is s_{j-1}, where $j = n_4 - n_3 + 1$. n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and j copies of symbol $*$.
  - Otherwise:
    End state is s_{i+1}. n_1 copies of symbol r_{i_1}, n_3 copies of symbol r_{i_3} and $n_4 + 1$ copies of symbol $*$.

- **Description:** Rules 1, 2 and 3 check conditions $r_{i_1} > k_i$, $r_{i_1} = k_i$ and $r_{i_1} < k_i$, respectively. If the condition of rule 2 is met, then rules 4 and 5 produce j copies of symbol $*$, where $j = n_4 - n_3 + 1$. Then, the j copies of symbol $*$ are used by the "GOTO" state rules to guide the cell transits to state s_{j-1}. If the condition of rule 2 is not met, then rule 1 or rule 3 prompts the cell to transit to state s_{i+1} and rule 6 produces one additional copy of symbol $*$.

3.4 Evolution rules for a READ instruction

The rules for an i-th instruction of the form (READ r_{i_1}) are as follow.

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_4 \geq 1$ copies of symbol $*$ and $n_5 \geq 1$ copies of symbol ϕ.
- **Rules:**

  1. $s_i \ * \rightarrow_{\texttt{min}} s_{i+1} \ * \ *$
  2. $s_i \ r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}$
  3. $s_i \ \phi \rightarrow_{\texttt{max}} s_{i+1} \ \phi \ r_{i_1}$

- **Postcondition:** End state is s_{i+1}. n_5 copies of symbol r_{i_1}, $n_4 + 1$ copies of symbol $*$ and n_5 copies of symbol ϕ.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 consumes all existing copies of symbol r_{i_1}. Rule 3 rewrites every copy of symbol ϕ into one copy of symbol ϕ and one copy of symbol r_{i_1}.

3.5 Evolution rules for a `HALT` instruction

There are no evolution rules for the last instruction `HALT`. If a system enters the state s_{n-1}, where n denote the number of instructions, then the system will halt.

3.6 Evolution rules for functional instructions

3.6.1 Rules for an i-th instruction of the form (`CALL` f r_{i_1} r_{i_2})

In the rule 1 below, multiset $x = \{r_i \mid r_i \text{ is a register of } M\}$, which can be determined when translating a given register machine M into the corresponding active P system Π_M.

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_2 \geq 1$ copies of symbol r_{i_2}, $n_4 \geq 1$ copies of symbol $*$ and $n_5 \geq 0$ copies of symbol $+$.
- **Rules:**

  Rules of state s_i:

  1. $s_i * \rightarrow_{\texttt{min}} s'_i \; * * \; [s_f, *^{f+1} \; x]$
  2. $s_i + \rightarrow_{\texttt{max}} s'_i$
  3. $s_i \; r_{i_1} \rightarrow_{\texttt{max}} s'_i \; r_{i_1} \; (+, \downarrow)$
  4. $s_i \; r_{i_2} \rightarrow_{\texttt{max}} s'_i$

  Rules of state s'_i:

  5. $s'_i + \rightarrow_{\texttt{max}} s_{i+1} \; r_{i_2}$

- **Postcondition:** End state is s_{i+1}. $n_4 + 1$ copies of symbol $*$, n_1 copies of symbol r_{i_1}, $n'_2 \geq 1$ copies of symbol r_{i_2}, where n'_2 is the number of symbol $+$ received from the current cell's children.
- **Description:** Rule 1 produces one additional copy of symbol $*$. At the same time, rule 1 creates one child cell with initial state s_f and initial content $\{*^{f+1} \; r_i \mid r_i \text{ is a register of } M\}$. Rule 2 consumes all existing copies of symbol $+$, if any, such that, later, the number of symbol $+$ received in state s'_i corresponds to the returned value from the function call made to the child. Rule 3 passes the parameter value, i.e. content of the register r_{i_1}, to the child by sending n_1 copies of symbol $+$ to the child. Rule 4 consumes all copies of symbol r_{i_2}. Rule 5 rewrites every copy of symbol $+$, received from the child, into one copy of symbol r_{i_2}, i.e. processes the returned value from the function call made to the child.

3.6.2 Rules for an i-th instruction of the form (`FUNC` f r_{i_1})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}, $n_4 \geq 1$ copies of symbol $*$ and $n_5 \geq 1$ copies of symbol $+$.
- **Rules:**

  1. $s_i * \rightarrow_{\texttt{min}} s_{i+1} \; * \; *$
  2. $s_i \; r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}$
  3. $s_i + \rightarrow_{\texttt{max}} s_{i+1} \; r_{i_1}$

- **Postcondition:** End state is s_{i+1}. n_5 copies of symbol r_{i_1}, $n_4 + 1$ copies of symbol $*$ and zero copies of symbol $+$.
- **Description:** Rule 1 produces one additional copy of symbol $*$. Rule 2 consumes all existing copies of symbol r_{i_1}. Rule 3 rewrites n_5 copies of symbol $+$ into n_5 copies of symbol r_{i_1}.

3.6.3 Rules for an i-th instruction of the form (`RETURN` r_{i_1})

- **Precondition:** $n_1 \geq 1$ copies of symbol r_{i_1}.
- **Rules:**
    1. $s_i\ r_{i_1} \rightarrow_{\texttt{max}} s_{i+1}\ (+, \uparrow)$
- **Postcondition:** End state is s_{i+1}. Zero copies of symbol r_{i_1}. n_1 copies of symbol $+$ sent to the current cell's parents.
- **Description:** For each copy of symbol r_{i_1}, rule 1 consumes the symbol r_{i_1} and sends one copy of symbol $+$ to the parent cells.

4 Conclusions

In this paper, we presented the details for constructing an active P system that simulates the functional register machines. Functional register machines of [1], extend the register machines of [2] by including instructions that: (i) define a function, (ii) make a function call and (iii) retrieve a return value from a function call.

Components of functional register machines are modeled as follows. Registers are represented by symbols, register contents are represented by multiplicity of the corresponding symbol, instruction lines are represented by cell states, instructions are represented by evolution rules.

Each constructed active P system has the following properties: (i) the number of states and evolution rules are proportional to the number of instructions of a given register machine and (ii) for each register machine instruction, the number of P system steps required to execute the corresponding rules is constant.

Acknowledgment

We want to thank Cristian Calude for suggesting this topic of study. This paper was supported by the Quantum Computing Research Initiatives at Lockheed Martin.

References

[1] C. S. Calude and D. Desfontaines. Anytime algorithms for non-ending computations. In preparation, The University of Auckland, 2014.

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

[3] G. J. Chaitin. *Algorithmic Information Theory.* Cambridge University Press, Cambridge, UK, 1987.

[4] E. Csuhaj-Varjú, M. Margenstern, G. Vaszil, and S. Verlan. On small universal antiport P systems. *Theor. Comput. Sci.*, 372(2-3):152–164, 2007.

[5] M. J. Dinneen and Y.-B. Kim. A new universality result on P systems. Report CDMTCS-423, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, Auckland, New Zealand, July 2012.

[6] R. Freund, L. Kari, M. Oswald, and P. Sosík. Computationally universal P systems without priorities: two catalysts are sufficient. *Theor. Comput. Sci.*, 330(2):251–266, 2005.

[7] M. Ionescu, G. Păun, and T. Yokomori. Spiking neural P systems. *Fundam. Inform.*, 71(2-3):279–308, 2006.

[8] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theor. Comput. Sci.*, 296(2):295–326, 2003.

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

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

[11] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Inc., New York, NY, USA, 2010.

[12] M. Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.