

# Computing Recursive Functions with P Systems (Extended Abstract)

Apostolos SYROPOULOS, Stratos DOUMANIS,  
Konstantinos T. SOTIRIADES

Greek Molecular Computing Group  
366, 28th October Str.  
GR-671 00 Xanthi, Greece, EU  
E-mail:gmcg@araneous.gr

## Abstract

P systems are conceptual computing devices that are at least as powerful as Turing machines. However, until recently it was not known how one can encode an arbitrary recursive function as a P system. Here we propose a new encoding of recursive functions as P systems with graph-like structure, which is the main difference with the previously documented attempts. The consequence of these efforts is that they provide a solid ground for the implementation of real programming languages in existing hardware.

## 1 Introduction

Classically intractable and unsolvable problems are the driving force behind the ever increasing search for new computational models. These new models can be used to provide faster solutions to classically intractable problems [6] and, in extreme cases, they can be *used* to *solve* classically unsolvable problems [5]. A major drawback of such new models is that they seem to be useful only for the solution of a particular class of problems. For example, DNA computing has proved efficient for the solution of combinatorial problems. Thus, a new computational model is really useful once we have the techniques that allow us to compute *ordinary* things.

P systems are a new model of computation inspired by the way cells live and function (see [1] for an overview of the theory). Although, in general, it has been proved that one can compute with P systems whatever (universal) Turing machines are capable to compute, until recently it was not clear how one can compute primitive recursive functions with P systems [8].

In this note we provide an alternative encoding of the basic functions as well as three processes, which can be applied to the basic functions to yield any  $\mu$ -recursive function. In order to encode these functions and processes we had to introduce new rewriting rules and to make use of a more “liberal” membrane structures. Thus, our encoding has stretched to the limit the capabilities of P systems as far it regards numerical computations.

In what follows, we present a very brief overview of the theory of P systems. Next, we present our encoding and we conclude with some remarks concerning our work and possible applications of it.

## 2 Brief Overview of P Systems

In what follows we will present an informal definition of P systems.

**Definition 2.1** A P system is a tuple

$$\Pi = (O, \mu, w_1, \dots, w_n, R_1, \dots, R_m, i_0),$$

where:

- (i)  $O$  is an alphabet (i.e., a set of distinct entities) whose elements are called *objects*.
- (ii)  $\mu$  is a membrane structure whose membranes are injectively labeled with succeeding natural numbers starting with one.
- (iii)  $w_i, 1 \leq i \leq m$ , are strings that represent multisets over  $O$  associated with each region  $i$ .
- (iv)  $R_i, 1 \leq i \leq m$ , are finite sets of rewriting rules (called *evolution rules*) over  $O$ . An evolution rule is of the form  $u \rightarrow v, u \in O^+$  and  $v \in O_{\text{tar}}^+$ , where  $O_{\text{tar}} = O \times \text{TAR}, \text{TAR} = \{\text{here, out}\} \cup \{\text{in}_j \mid 1 \leq j \leq m\}$ .
- (v)  $i_0 \in \{1, 2, \dots, m\}$  is the label of an elementary membrane (i.e., a membrane that does not contain any other membrane), called the *output* compartment.

Note that the keywords “here,” “out,” and “in<sub>*j*</sub>” are called *target* commands. Given a rule  $u \rightarrow v$ , the length of  $u$  (denoted  $|u|$ ) is called the radius of the rule. A P system that contains rules of radius greater than one is a system with cooperation.

The rules that belong to some  $R_i$  are applied to the objects of the compartment  $i$  synchronously, in a non-deterministic maximally parallel way. Given a rule  $u \rightarrow v$ , the effect of applying this rule to a compartment  $i$  is to remove the multiset of objects specified by  $u$  and to insert the objects specified by  $v$  in the regions designated by  $v$  in the target commands associated with the objects from  $v$ . In particular,

- (i) if  $(a, \text{here}) \in v$ , the object  $a$  will be placed in compartment  $i$  (i.e., the compartment where the action takes place),
- (ii) if  $(a, \text{out}) \in v$ , the object  $a$  will be placed in the compartment that surrounds  $i$ ,
- (iii) if  $(a, \text{in}_j) \in v$ , the object  $a$  will be placed in compartment  $j$ , provided that  $j$  is immediately inside  $i$ , or else the rule is not applicable.

### 3 Encoding the Basic Functions as P Systems

Our exposition on recursion theory is based on standard references [4, 3]. Here are the three basic functions:

- (i) the *successor* function  $S(x) = x + 1$ ,
- (ii) the *zero* function  $z(x_1, \dots, x_n) = 0$ , and
- (iii) the *identity* function  $U_i^n(x_1, \dots, x_n) = x_i, 1 \leq i \leq n$ .

If we want to define  $A$ -recursive functions, we also need to use the characteristic function. However,  $A$ -recursive sets are not important for our discussion, so we are going to restrict ourselves to general recursive functions only.

Two nested compartments (i.e., one inside another) make up the simplest non-trivial membrane structure. For this reason, we have opted to encode the basic functions as P systems that have this particular membrane structure. In what follows, the inner compartment will be the target compartment. Let us now elaborate on the encoding of each function.

**The zero function** This function simply discards its arguments and returns the number zero. For each argument  $x_i$  of the function we pick up an object  $\alpha_i$  and place  $x_i$  copies of it in the outer compartment. Next, we associate to each object  $\alpha_i$  a multiset rewrite rule of the form

$$\alpha_i \rightarrow \varepsilon$$

This is a new kind of rule that simply annihilates all objects as it replaces each object with the empty string. We can imagine that there is a pipe that is used to throw the  $\alpha_i$ 's into the environment. This rule can be considered to implement a form of “catharsis” of a compartment. Thus, the outcome of the computation of this P system is the number zero, hence this system implements the zero function.

**The successor function** Suppose we have a P system with no data that, at the same time, encodes the successor function; then, this system must place an object to the target compartment and stop. Obviously, the problem is how this can happen, as there is no rule that can be applied to an empty compartment. For this reason, we introduce the new rule

$$\varepsilon \rightarrow (\alpha, \text{in}_i)$$

This rule can be applied only if there are no objects in a given compartment. And its effect is to place an object  $\alpha$  to compartment  $i$ . After this, the rule is “disassociated” from the current compartment. In other words, this rule can be used only once. The rationale behind the introduction of this rule is that at any moment a cell can absorb matter from its environment that, in turn, will be consumed. Having this new rule, it is now trivial to encode the successor function.

**The identity function** This function is similar to the zero function with one difference: it discards all of its arguments but one. So, for each argument  $x_i$  we place  $x_i$  copies of an object  $\alpha_i$ . Note that all objects  $\alpha_i$  are distinct. Assume that the particular instance of the identity function returns its  $j$ th argument, then we associate with  $\alpha_j$  the following rule:

$$\alpha_j \rightarrow (\alpha_j, \text{in}_2)$$

Similarly to the zero function case, all other objects are associated with a catharsis rule:

$$\alpha_i \rightarrow \varepsilon, i \neq j$$

## 4 Encoding the Processes as P Systems

Recursive functions can be defined by applying function builders, or processes, to the basic functions. For example, the addition function is build by applying primitive recursion to the identity and successor functions. Thus, it is necessary to provide an encoding of the three processes in order to be able to compute any recursive function using P systems. The three processes are described below:

**Composition** Suppose that  $f$  is a function of  $m$  arguments and each of  $g_1, \dots, g_m$  is a function of  $n$  arguments, then the function obtained by composition from  $f, g_1, \dots, g_m$  is the function  $h$  defined as follows:

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

**Primitive Recursion** A function  $h$  of  $k + 1$  arguments is said to be definable by (primitive) recursion from the functions  $f$  and  $g$ , having  $k$  and  $k + 2$  arguments, respectively, if it is defined as follows:

$$\begin{aligned} h(x_1, \dots, x_k, 0) &= f(x_1, \dots, x_k) \\ h(x_1, \dots, x_k, S(m)) &= g(x_1, \dots, x_k, m, h(x_1, \dots, x_k, m)) \end{aligned}$$

**Minimization** The operation of minimization associates with each total function  $f$  of  $k + 1$  arguments the function  $h$  of  $k$  arguments. Given a tuple  $(x_1, \dots, x_k)$ , the value of  $h(x_1, \dots, x_k)$  is the least value of  $x_{k+1}$ , if one such exists, for which  $f(x_1, \dots, x_k, x_{k+1}) = 0$ . If no such  $x_{k+1}$  exists, then its value is undefined.

In the rest of this section we will not provide encodings for the general case. Instead, we will concentrate on simple cases that can be easily generalized. To put it in another way, if the P systems that we present faithfully encode the basic processes, then one can easily generalize the encoding scheme.

**Encoding the composition process** Assume that  $f$  is a function that has two arguments and that  $g_1$  and  $g_2$  are two functions with three arguments each, then we want to design a P system that will encode a function  $h$  defined as follows

$$h(x_1, x_2, x_3) = f(g_1(x_1, x_2, x_3), g_2(x_1, x_2, x_3)) \quad (1)$$

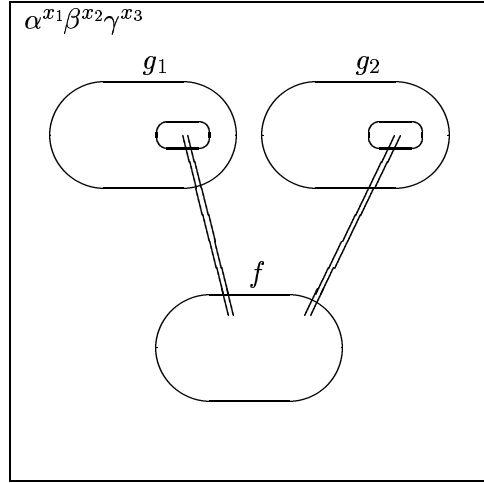


Figure 1: A P system encoding the composition process.

Given three integers  $x_1$ ,  $x_2$  and  $x_3$ , we want to design a P system capable to compute  $h(x_1, x_2, x_3)$ . From equation 1 it is clear that we have first to compute  $u_1 = g_1(x_1, x_2, x_3)$  and  $u_2 = g_2(x_1, x_2, x_3)$ . This suggests that the P system will contain two nested P systems that will compute the numbers  $u_1$  and  $u_2$ , respectively. In addition, these numbers (i.e., a multiset of objects representing them) have to be directly supplied to a third nested P system, which will compute the value  $u = f(u_1, u_2)$ . This means that we need two pipelines that will connect the output compartment of the first two nested P systems with the P system that will compute the number  $u$ . Obviously, P systems with tree-like structure are not adequate to solve this problem. This means that we need a way to more generally specify the communication channels between the various compartments. An obvious solution, is to use P systems that have a (limited) graph-like structure. Indeed, in [7] the first author of this paper provides such an extension.

**Definition 4.1** A graph-structured symbol-object membrane system is a set of (possibly nested) compartments that can be specified as follows:

$$\Pi = (O, g_m, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$$

Here:

- (i)  $O$  is a set of objects that denote elements of information;
- (ii)  $g_m$  is a relation in  $\{1, \dots, m\} \times \{1, \dots, m\}$ , describing the network structure of the system;
- (iii)  $w_i$ ,  $1 \leq i \leq m$ , are strings representing multisets of objects that are elements of  $O$  that denote conglomerations of information;

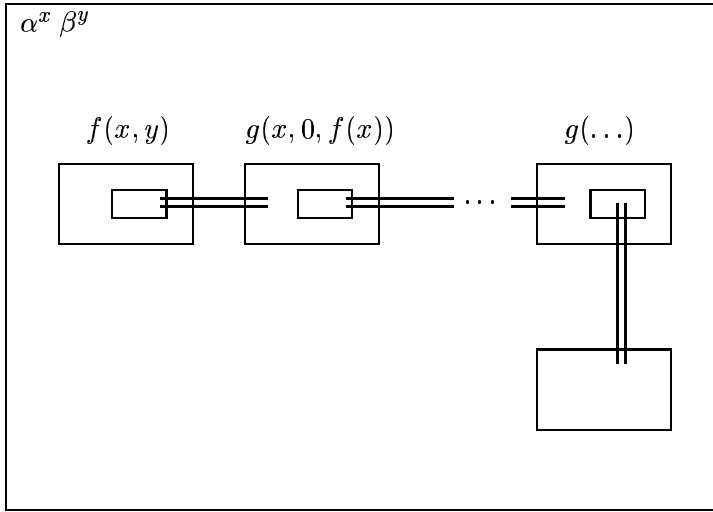


Figure 2: A P system encoding primitive recursion.

- (iv)  $R_i$ ,  $1 \leq i \leq m$ , are finite sets of *evolution rules* over  $O$ ;  $R_i$  is associated with compartment  $i$ ; an evolution rule is of the form  $u \rightarrow v$ , where  $u$  is a string over  $O$  and  $v$  is a string over  $O_{\text{NTAR}}$ , where  $O_{\text{NTAR}} = O \times \text{NTAR}$ , for  $\text{NTAR} = \{\text{here, out}\} \cup \{\text{to}_j | 1 \leq j \leq m\}$ ;
- (v)  $i_0 \in \{1, 2, \dots, m\}$  is the label of the *output compartment*.

The symbol “to <sub>$j$</sub> ” should be used to directly place an object from the host compartment to compartment  $j$ . The rule is applicable only if  $(i, j) \in g_m$  or  $(j, i) \in g_m$ .

**Encoding primitive recursion** In primitive recursion we know a priori the number of iterations. Thus, if we want to compute  $h(x, y+1)$  we actually have to compute the following:

$$g(x, y, g(x, y - 1, g(x, y - 2, \dots g(x, 0, f(x)) \dots)))$$

Thus, if we want to compute  $h(x, y)$ , we need  $n$  compartments that will compute the successive values of  $g$  and one compartment that will compute the value of  $f(x)$ . In addition, all these compartments must form a pipeline where the  $i$ th compartment will get its data from compartment  $i - 1$  and it will feed its output to the compartment  $i + 1$ . Naturally, the last compartment will populate with data the output compartment. Initially, all compartments will get the same copies of a designating object, say #, which will denote the first argument of function  $h$ . Also, the compartments that compute the successive values of  $g$  will get the respective number of copies of another designated object, say @. Figure 2 depicts the initial setting of the system.

**Encoding the minimization function builder** As for the previous cases, we present a simplified system that computes function  $h(x)$ . Assume that  $f$  is a function

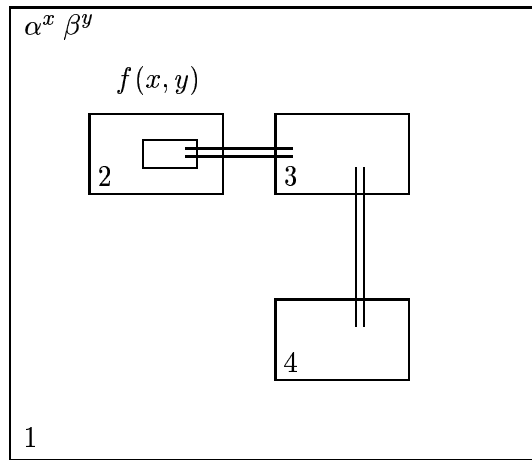


Figure 3: A P system encoding minimization function.

of two arguments  $x$  and to compute  $f(x, y)$  with a fixed  $x$ , which is equal to the argument of the original function  $h$ , and a variable  $y$ , which is increased by one in each iteration, that halts the computation the very moment the outcome of the function computation is zero. Figure 3 shows the general setup of a P system that computes function  $h(x)$ . Compartment 3 is directly connected to the output compartment of compartment 2, where actual the computation is taking place. The output of this compartment is pipelined to compartment 3 and it is transformed to the distinguished object  $\#$ . This symbol is used as counter. The connection from the compartment number 3 to compartment number 4 is used to move the object  $\#$  to it. In addition, at each iteration, the  $a$ 's and  $b$ 's are also placed in compartment 3 in the form of of two distinguished elements  $\oplus$  and  $\otimes$ . More specifically, we move the  $a$ 's and the  $b$ 's in compartment 1 and at the same time we create a number of  $\oplus$ 's and  $\otimes$ 's that is to number of  $a$ 's and  $b$ 's respectively. Clearly, if the result of the computation at compartment 2 is the number zero, no action should take place at compartment 3. Otherwise, we first move  $\#$  to compartment 4 and place a copy of  $b$  at the compartment 1, then we place the  $\oplus$ 's and  $\otimes$  to compartment 1 in the form of  $a$ 's and  $b$ 's. This way, we actually increase the  $b$ 's by one, which is what we actually need to do to get the correct result.

It is important to note that in all cases our description is a little bit vague, as we do not give the full details of the encoding. However, our intension was to give the general idea of how things can be done. After all, we believe it is not difficult to work out these details so to provide a complete description of the encoding of each function builder.

## 5 Conclusions

We have presented an alternative encoding of the basic functions and the three function builders of recursion theory. Since any functional programming language can be used to compute any recursive function, one can build a front-end capable of compiling (simple) functional programs into P systems. Indeed, in [7], the first author of this paper presents ideas related to this particular implementation procedure. We believe that more work is needed to be done in order to provide a complete and viable solution to the problem of encoding recursive functions as P systems. We just hope that our work is a step towards this direction.

## References

- [1] Gh. Păun, *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, 2000.
- [2] Gh. Păun, *Computing with Membranes*, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
- [3] M. Davis, *Computability and Unsolvability*, Dover Publications, Inc., New York, 1982
- [4] G.S. Boolos and J.P. Burgess and Richard C. Jeffrey, *Computability and Logic*, Cambridge University Press, Cambridge, UK, fourth edition, 2002.
- [5] T.D. Kieu, *Quantum Hypercomputation*, *Minds and Machines*, 12 (2002), 541–561.
- [6] L. Adleman, *Molecular Computation of Solutions to Combinatorial Problems*, *Science*, 266 (1994), 1021–1024.
- [7] A. Syropoulos, *On P Systems and Distributed Computing*, To appear in the pre-proceedings of WMC5, 2004.
- [8] A. Romero-Jiménez, M. J.Pérez-Jiménez, *Computing Partial Recursive Functions by Transition P Systems*, in Martín-Vide, Carlos and Gheorghe Păun, eds., *Membrane Computing*, LNCS 2933, Springer-Verlag, Berlin, 2004, 320–340.