CS692 Report : P systems using worms

Avadhut Sardeshmukh
Roll No 06329905
Under the guidance of Prof. Krishna S.
Computer Science and Engineering

May 5, 2008

Abstract

We study the variant of P systems where the objects are symbols as well as strings (a.k.a. worms). They deal with multisets of strings and the result of a computation is the number of strings in the output membrane, as usual. It has been proved that membrane systems (using worms) with at most 6 membranes are universal. Also P systems using worms for some specific languages are constructed and presented as illustrative examples. Finally a linear-time solution of the the SAT problem, from [1] is examined and a uniform solution for the same is obtained.

1 Introduction

Here we consider the type of P systems in which the objects are symbols as well as strings. The strings are also called as worms. The rules in this type of systems deal with strings (symbol objects are treated as strings of unit length). These type of P systems were introduced in [2]. The result of computation is the number of strings in the output membrane. The strings are processed by four types of operation, viz. replication, splitting, recombination and mutation. Replication and splitting can increase the number of string objects whereas recombination and mutation cannot. These operations are discussed in detail in the following sections.

1.1 Formal Definition

P System with worms:
[Definition] A P system of degree $m \geq 1$ with worm-objects is a construct :

$$\Pi = (V, \mu, A_1, \ldots, A_m, (R_1, S_1, M_1, C_1), \ldots, (R_m, S_m, M_m, C_m), i_0)$$

where

- *V is an alphabet*
- *μ is a membrane structure consisting of m membranes arranged in an heirarchical structure*
- *A_i are finite multisets over V^*, associated with the corresponding regeions of the structure μ.*
- *R_i, S_i, M_i, C_i are finite sets of replication, splitting, mutation rules and the set of objects used as crossing over blocks in the recombination operation. The forms of these rules as well as the operations they do are explained in the next section.*
- *$i_0 \epsilon \{1, 2, \ldots, m\}$ is the output membrane*

1.2 Operations

1. Replication $r : (a \rightarrow u_1||u_2; tar_1, tar_2)$ or $(a \rightarrow u_1||u_2; tar_1, tar_2)\delta$, where $a \epsilon V,\ u_1, u_2 \epsilon V^+$. This type of rule can be applied on a string w of the form $x_1 a x_2$. It results in two strings, by replacing the occurrance of a in w by u_1 and u_2, respectively. We say that $w_1 \Longrightarrow_r (w_2, w_3)$, if $w_1 = x_1 a x_2$, $w_2 = x_1 u_1 x_2$ and $w_3 = x_1 u_2 x_2$. After application of this rule, the strings w_2 and w_3 are sent to the membranes indicated by tar_1 and tar_2, respectively. The optional δ at the end of the rule specifies the action of dissolving the membrane. If a rule of that form is used, the membrane is dissolved at the end of that step. (We will discuss what this means in detail later).

2. Splitting $r : (a \rightarrow u_1 : u_2; tar_1, tar_2)$ or $(a \rightarrow u_1 : u_2; tar_1, tar_2)\delta$ where $a \epsilon V,\ u_1, u_2 \epsilon V^+$. Again this type of rule can be applied on a string w of the form $x_1 a x_2$. It results in two strings – obtained by cutting

w at a, and replacing the a by u_1 in one part and by u_2 in the other part. That is, we say $w_1 \Longrightarrow_r (w_2, w_3)$, if $w_1 = x_1 a x_2$, $w_2 = x_1 u_1$ and $w_3 = u_2 x_2$. Also, the strings w_2 and w_3 obtained in this way are sent to membranes specified by tar_1 and tar_2, respectively, as before. Again the optional δ at the end specifies membrane dissolution action.

3. Mutation $r : (a \rightarrow u; tar)$ or $(a \rightarrow u)\delta$. This is a context free rule. That is, for a string containing a, this rule can be applied to obtain a different string by replacing a by u. We say $w_1 \Longrightarrow_r w_2$, if $w_1 = x_1 a x_2$, $w_2 = x_1 u x_2$. The string w_2 thus obtained, is sent to the membrane specified by tar, as before. And δ is the membrane dissolution action.

4. Recombination $r : (z; tar_1, tar_2)$ or $(z; tar_1, tar_2)\delta$. This type of rule can be applied to a pair of strings (say w_1 and w_2) which have z as their common substring. The operation involvs "splicing" the strings w_1 and w_2 at z and recombining their splices in a crossed manner. That is we say $(w_1, w_2) \Longrightarrow_r (w_3, w_4)$ if $w_1 = x_1 z x_2$, $w_2 = y_1 z y_2$ and $w_3 = x_1 z y_2$, $w_4 = y_1 z x_2$ As before, the strings w_3 and w_4 are sent to membranes specified by tar_1 and tar_2, respectively. And δ means the membrane is dissolved after the current step.

Note that in all four type of rules above, the strings x_1, x_2, y_1, y_2 ϵV^*. Note also that mutation rules can delete symbols because the string u ϵ V^*. Also, replication and splitting rules can increase the total numbers of worms/strings, but recombination and mutation cannot.

We will now see what the membrane dissolution operation means and how does a P system with worms evolve.

1.3 Principles for evolution

The $(m+1)$-tuple $(\mu, A_1, \ldots, A_m)$ constitutes the initial configuration of the system. Starting with this, the system can pass from one configuration to another according the following principles :

- The work of the system is synchronized. That is, in each step, in each region, all strings that can be processed by a rule in that region are processed. This is called maximal parallel approach. Also, if more than one rules are applicable to a particular string (or a rule is applicable to more than one string) then the choice is done non-deterministically. So,

the strings in all membranes are assigned to rules in their respective membranes in a non-deterministic, maximally parallel manner. And after the assignment is complete, the system evolves one step.

- A particular object (i.e. a particular copy of a string) can be processed by only one rule at any time, although different copies of the same string can be assigned to different rules, if possible.

- The strings resulting from the various operations are sent to the membranes specified by the targets in that rule. *here* means the string is not moved anywhere. *out* means that the resulting string is to be sent to the region surrounding the one in which this rule is applied. Simillarly, in_j means the string is sent inside, to the membrane numbered j, provided, that membrane is adjacent to the present membrane. That is, we can only send strings between adjacent regions.

- When a rule containing the symbol δ is applied, the current membrane is dissolved. This means that the objects in this membrane are left free in the membrane surrounding it, and the rules of this membrane will be lost. The skin membrane can never be dissolved. In the event of a membrane dissolution, first all objects evolve according to the rules assigned to them (and may be are sent to other membranes also) and then the membrane is dissolved. For example, if during a step, a membrane i has objects $\{(a,1),(bb,2),(c,1)\}$ and rules $(a \rightarrow f\ ; out)$ and $(b; here, in_j)\delta$ then a evolves to f, is sent out of membrane i; the two copies of bb are recombined using the second rule, one is kept here and the other sent to membrane j, and then membrane i is dissolved.

2 Examples

2.1 Example 1 : Generating numbers of the form $m * n$

This is a simple P system for computing multiplication of two numbers. The P system is independent of the two numbers to be multiplied. The set of numbers generated by it is the set of numbers of the form $m * n$. Consider the P system defined as follows :

$$\Pi = (V, \mu, A_1, \ldots, A_6, (R_1, S_1, M_1, C_1), \ldots, (R_6, S_6, M_6, C_6), 3)$$

Where
$\mu = [_1[_2[_3]_3[_4]_4[_5]_5[_6]_6]_2]_1$,
$V = \{a, b, c\}$,
$A_5 = \{(a, 1)\}$, $A_6 = \{(b, 1)\}$ and $A_i = \phi, 1 \leq i \leq 4$,

$R_1 = \{(a \rightarrow a||a; here, here)\}$, $S_1 = M_1 = C_1 = \phi$

$R_2 = \{(c \rightarrow c||c; here, in_3)\}$, $M_2 = \{(a \rightarrow \lambda; here)\delta\}$,
$C_2 = \{(a; here, in_4)\}$, $S_2 = \phi$

$R_3 = S_3 = M_3 = C_3 = R_4 = S_4 = M_4 = C_4 = \phi$

$R_5 = \{(a \rightarrow a||a; here, here), (a \rightarrow a||a; here, here)\delta\}$, $S_5 = M_5 = C_5 = \phi$
$R_6 = \{(b \rightarrow b||c; here, here), (b \rightarrow b||c; here, here)\delta\}$, $S_6 = M_6 = C_6 = \phi$

Evolution. The work of this system is very straight forward. In membrane 5 and membrane 6, it generates the input numbers (i.e. the multiplicand and the multiplier) by using replication rules. At any point, there membranes can be dissolved. Note that the two membranes can be dissolved independantly, so the final result may not be the multiplication of these numbers. We shall see how.

Once the membranes 5 and 6 are dissolved (assume, for the time being, that they dissolve at the same time), we have 2^n a's and m c's in membrane 2. Now the recombination of pairs of a's starts happening in membrane 2. Everytime, half of the a's are thrown in membrane 4. So in n steps, there remains only one a. Till this time, in each step, m c's are copied in membrane 3 – the output membrane. This last a is now consumed by the operation $(a \rightarrow \lambda)\delta$ and the membrane dissolves. The computation halts at this time and we get $m * n$ number of c's in output membrane. Now, in membrane 2, if the rule $(a \rightarrow \lambda)\delta$ is used at any other time than the last (i.e. other than what is intended), then at least two copies of a arrive in membrane 1 and the system loops forever by the recombination rule there.

Now, what happens if both membranes do not dissolve at the same time? Suppose that membrane 5 dissolves first. Then by the time membrane 6 dissolves, membrane 2 would have lost some of the 2^n a's that it got. Lets say the number of a's becomes $2^{n-n'}$, then we will end up calculating $(n-n') * m$. A similar argument applies to the case of membrane 6 getting dissolved first. Finally, we calculate a number of the form $m * n$.

2.2 Example 2 : Generating the length set of the language $\{a^{n!}\}$

Here we give a nonuniform solution. That is, the number of membranes in this P system will be dependent on the factorial number that you want to generate. Suppose we want to compute $n!$. Consider the following P system with $n+2$ membranes.

$$\Pi = (V, \mu, A_1, \ldots, A_{n+1}, (R_1, S_1, M_1, C_1), \ldots, (R_{n+1}, S_{n+1}, M_{n+1}, C_{n+1}), n+2)$$

Where
$V = \{b_i | 0 \leq i \leq n\} \cup \{x_i | 1 \leq i \leq n\}$,
$\mu = [_{n+1}[_n \ldots [_1]_1 \ldots]_n[_{n+2}]_{n+2}]_{n+1}\}$,
$A_1 = \{(b_0, 1)\}$, $A_i = \phi$, $2 \leq i \leq n+2$,

For all i such that, $1 \leq i \leq n$, the sets of rules for membrane i are as follows :
$R_i = \{(b_{i-1} \rightarrow b_{i_1} || b_i; here, out)\}$ and
$M_i = \{(x_1 \rightarrow x_2; here), (x_2 \rightarrow x_3; here) \ldots (x_{n-i} \rightarrow x_{n-i+1}; here),$
$(x_{n-i+1} \rightarrow \lambda)\delta\}$

Finally, the rules for membrane $n+1$ are :
$M_{n+1} = \{(b_n \rightarrow b_n; in_{n+2})\}$
There are no other rules in the system.
Evolution. How does this P system work? It has initially 1 b_0 in membrane 1. From membrane i, it sends out as many b_i's as $n-i+1$ times the number of b_{i-1}'s that it has. This is because, in membrane i we count from 1 to $n-i+1$ by using the objects x_i and till then, we keep sending as many b_i's out, as the number of b_{i-1}'s we have. Hence, n b_1's are sent out of membrane 1. Then that is multimplied by $n-1$ (put $i=2$ in $n-i+1$) in membrane 2. This quantity is multiplied by $n-3$ in membrane 3 (similarly) and so on.

Finally, in membrane $n+1$ we get number of b_n's equal to $n!$, which are then sent to the output membrane by the rule $(b_n \rightarrow b_n; in_{n+2})$.

2.3 Example 3 : Generating the Fibonacci numbers

Consider the P system Π defined as follows :

$$\Pi = (V, \mu, A_1, A_2, A_3, (R_1, S_1, M_1, C_1), (R_2, S_2, M_2, C_2), 1)$$

Where,
$V = \{f_1, f_2, f, x, y\}$,
$\mu = [_1[_2]_2[_3]_3]_1$,
$A_1 = \phi$, $A_2 = \{(f_1, 1)\}$, $A_3 = \{(f_2, 1), (x, 1)\}$
$R_1 = \{(f_2 \rightarrow f||f_1; here, in_1)\}$, $M_1 = \{(f_1 \rightarrow f; here), (f \rightarrow f_2; in_2), x \rightarrow y; in_1\}$,

$R_2 = \phi$, $M_2 = \{(f_1 \rightarrow f_1; out), (y \rightarrow \lambda; out)\delta\}$,

$R_3 = \phi$, $M_3 = \{(f_2 \rightarrow f_2; out), (f_2 \rightarrow f_2; out)\delta\}$,

$S_2 = S_3 = C_2 = C_3 = \phi$.
Evolution. Lets examine the evolution of this system. Its based on the simple recurrence to calculate fibonacci numbers : $f_n = f_{n-1} + f_{n-2}$. Membrane 2 stores objects of type f_1 and membrane 3 stores objects of type f_2 (initialy one each). Both of them send their objects as such to membrane 1 i.e. out. Membrane 1 is our output membrane. If the computaion halts at a step, it has number of f's equal to some n^{th} fibonacci number.

Initially membrane 2 and membrane 3 have one f_1 and one f_2 respectively. They send them to membrane 1, where the f_1's are mutated to f's straight away, whereas the f_2's are replicated as f's here (membrane 1) and as f_1's in membrane 2. All the f's in membrane 1 are now sent back to membrane 2, as f_2's. And the addition can continue. This is nothing but following the recurrence. In short if $n(x)$ denotes the current number of objects of type x in membrane 1, and $n'(x)$ denotes the new number (i.e. after this step), then $n(f) = n(f_1) + n(f_2)$, and then we put the new values as $n'(f_1) = n(f_2), n'(f_2) = n(f)$, so that we always add the most recent two numbers in the sequence.

At any time, we can decide to stop this computation by applying the rule $(f_2 \rightarrow f_2; out)\delta$, which dissolvs membrane 3. When this happens, the object x waiting there from the first step is set free in membrane 1, where in the next step itself the rule $(x \rightarrow y; in_2)$ is applied and the object y is produced in membrane 2. This y now causes membrane 2 to be dissolved by the rule $(y \rightarrow \lambda; out)\delta$ and the new number of f_1's – $n'(f_1)$ which is equal to $n(f_2)$ also adds to the current number of f's to give the next fibonacci number in sequence and the computation stops.

Observe that there is a choice to dissolve membrane 3 but dissolution of membrane 3 automatically triggers the dissolution of membrane 2 and stops

the computation after calculating one more fibonacci number in sequence.

3 Solving NP Complete problems using P systems with worms

3.1 A semi-uniform solution to SAT

Here we build upon the solution given in [1], to form a P system with 5 membranes to solve SAT.

Given a formula in CNF, SAT asks whether there exists an assignment of truth values to the variables which renders the formula to be true. The given formula is in the following form :

$$\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

where each C_i is of the form $x_1 \vee x_2 \vee \ldots \vee x_r$ such that $i \leq r \leq n$.

That is, there are n variables, and m clauses in all, and each clause has some r number of variables ORed together. All the clauses are ANDed. For the formula to be true, at least one variable from each of the clauses must be true. Note that the occurrances of x_i's in the clauses above can be either in the normal form or in the primed form (complimented), in which case a false assigned to that variable renders the clause (containing it) to be true. The problem here is there are 2^n possible truth assignments, and we have to check all of them. In [1], a P system with $m + 1$ membranes to solve SAT was constructed. The logic is, generate all possible assignments in the innermost membrane and then check if each clause is satisfied in every next surrounding membrane. Send out the string corresponding to an assignment out of membrane i, if and only if all clauses upto C_i have been made true by that assignment. Thus a string leaves the outermost membrane if and only if the formula is satisfiable. The same system can be used here by introducing an output membrane in the outermost membrane.

We here try to give a semi-uniform solution in that the number of membranes will be constant (namely, 5), irrespective of the size of the instance of SAT problem given.

Define the P system with worm objects Π of size 5 as follows :

$$\Pi = (V, \mu, A_1, \ldots, A_5, (R_1, S_1, M_1, C_1), \ldots (R_5, S_5, M_5, C_5), 5)$$

Where,
$V = \{x_i, t_i, f_i, t_{ij}, f_{ij} | 1 \leq i \leq n,\ 1 \leq j \leq m\} \cup \{X, Y, D, D_1, D_2, YES, \dagger\}$,
$\mu = [_1[_2]_2[_3[_4[_5]_5]_4]_3]_1$,
$A_1 = \{(X, 1)\},\ A_2 = \{(x_1, 1)\}$ and , $A_i = \phi,\ 3 \leq i \leq 5$,

$R_1 = \{(Y \rightarrow D || \dagger; here, here)\},\ M_1 = \{(D \rightarrow X; here)\}$,
$C_1 = \{(X; in_3, in_3), (Y; out, out), (\dagger; here, here)\},\ S_1 = \phi$

$R_2 = \{(x_i \rightarrow t_i x_{i+1} || f_i x_{i+1}; here, here) | 1 \leq i \leq n-1\}$
$\cup \{(x_n \rightarrow t_n X || f_n X; out, out)\}$,
$S_2 = C_2 = M_2 = \phi$,

$M_3 = \{(t_i \rightarrow t_{i1}; here) | 1 \leq i \leq n\}$
$\cup \{(f_i \rightarrow f_{i1}; here) | 1 \leq i \leq n-1\}$
$\cup \{(t_{ij} \rightarrow t_{ij+1}; here) | 1 \leq i \leq n\}$
$\cup \{(f_{ij} \rightarrow f_{ij+1}; here) | 1 \leq i \leq n-1, 1 \leq j \leq m\}$
$\cup \{(D \rightarrow D_1; here), (D_1 \rightarrow D_2; here), (D_2 \rightarrow Y; out)\}$
$\cup \{(t_{im} \rightarrow YES; in_5), (f_{im} \rightarrow YES; in_5)\}$,
$R_3 = \{(f_{n-1j} \rightarrow fnj || D; in_4, here) | 1 \leq j \leq m\}$,
$S_3 = C_3 = \phi$,

$R_4 = \{(t_{ij} \rightarrow t_{ij} || D_2; out) | x_i \epsilon C_j\} \cup \{(f_{ij} \rightarrow f_{ij} || D_2; out) |\ ^{-}x_i\ \epsilon C_j\}$,
$M_4 = S_4 = C_4 = \phi$,

$R_5 = M_5 = S_5 = C_5 = \phi$
Lets understand the work of the above system. As said above, the system first generates all possible truth assignments to the n variables and then checks if at least one of these assignments satisfies the given formula.

Membrane 2 initially has a single x_1 object. From there the replication rules keep on generating all possible assignments. Presence of t_i in the generated string means x_i is assigned the value TRUE and presence of f_i means the opposite. Thus, after n steps, all possible truth assignments are computed (each corresponds to a string of t_i's and f_i's) and they are sent to membrane 1 (out), by appending an X in the end.

The function of membrane 1 is to send these strings one by one into membrane 3 for checking if they satisfy the formula. For this, the recombination with X is used. There is only a single X initially in membrane 1. This recombines with any one of the strings and moves both X and that string in

membrane 5. Once a string is sent inside for checking, next string can only be sent when an X is produced/brought here, in membrane 1. So lets see what happens to the truth-assignment string sent inside membrane 3.

In membrane 3, the checking of whether each of the m clauses of the given formula evaluate to true with this particular assignment of truth values–i.e. the one represented by this string which has come in. This is done step by step for each clause. If any clause fails to be true for this assignment, an X is sent to the outside membrane (i.e. mem. 1) to get the next assignment in for checking. This is done as follows :
When the string of t_i's and f_i's enters first, all t_i's are mutated to t_{i1}'s, one by one, then all f_i's mutated to f_{i1}'s, again one by one (One by one because there is only one string, so at any time, only one operation/rule can process it). And with the last f_i converted to f_{i1}, the string enters into membrane 4. The subscript 1 attached indicates that clause 1 is being checked for truth. The string sent inside comes back if and only if this assignment makes clause 1 true (we'll see why). When it comes back, the subscript is increased and the string is again sent inside. If the string doesn't come back, it means this assignment does not satisfy the formula and we need to check next one. For this, the D, which was generated when the string was sent in, waits for two steps for the string to come back. And then, sents a single Y out to membrane 1, which evolves into X, as required. But if the string comes back, then 2 Y's are sent to membrane 1, instead of 1. In this case both the Y's are destroyed (otherwise if two †'s are produced by two Y's, then the system loops forever).

Lets see why the entered string comes out of membrane 4 if and only if the current clause is true w.r.t. the current assignment. Membrane 4 has only rules of the form $(t_{ij} \rightarrow t_{ij}; out)$, if literal x_i is present in clause C_j and $(f_{ij} \rightarrow f_{ij}; out)$ if literal $^{-}x_i$ is present in clause C_j. But currently, all variables in the string have got the subscript p (as in t_{ip}) if current clause being checked is C_p. So, only rules corresponding to C_p will have effect, if any. And only if at least one of the literals present in the C_p is assigned a value TRUE, the rules will be applicable. In case two or three are applicable, any one assigned does the trick. For example if $C_3 = x_1 \vee {}^{-}x_2 \vee x_3$ and a string of the form $t_{13}t_{23}t_{33}$ can be processed by the rule $(t_{13} \rightarrow t_{13}; out)$ or similar rule with t_{33}. Note that these rules will be present in membrane 4, according to our definitions, and all other rules will not have any effect, because they have different values of j than 3.
So if the clause is not satisfied under current assignment, the string can never

come out.

If the string never comes out, the Y (single) that is sent to membrane 1, produces X there and sends the next assignment for checking. Otherwise, the string that comes out, has a D_2 in it. So one single Y and a string containing Y are sent to membrane 1. There, the 2 Y's can either be recombined and destroyed or the they can produce two X's, in which case they also produce two †'s, which trap the system. Hence to continue, we must destroy the Y's.

Finally, if all m clauses are rendered true by a certain assignments, then the symbols t_{1m} or f_{1m} produced in membrane 4, send an object of type YES to the output membrane.
So, if the output of the system is non-zero, then the formula is satisfaible, otherwise its not.

4 Conclusion and future work

The solutions given here can be improvised in many ways. For example, the P system given for generating the factorials is dependent on n. A uniform solution in this case will generate the language $\{a^{n!}\}$. Also, the universality of these type of P systems remains to be investigated. Universality has been obtained with 6 membranes [3]. It is still an open problem to prove a lower bound on the number of membranes required for computational completeness. Finally, the uniform solution to SAT given here proves that P systems with worms are a good candidate for solving NP-complete problems. A solution to HPP can be obtained on the similar lines. One such solution from [2] was studied.

References

[1] S.N.Krishna. *Languages of P systems : Computabiliy and Complexity.* PhD thesis, IIT Madras, 2001.

[2] Gheorghe Paun. Computing with membranes: P systems with worm-objects. 2000.

[3] Gheorghe Paun. Computing with membranes: One more collapsing hierarchy.