# A CIRCUIT IMPLEMENTING MASSIVE PARALLELISM IN TRANSITION P SYSTEMS

Santiago Alonso, Luis Fernández, Fernando Arroyo, Javier Gil

***Abstract***: *Transition P-systems are based on biological membranes and try to emulate cell behavior and its evolution due to the presence of chemical elements. These systems perform computation through transition between two consecutive configurations, which consist in a m-tuple of multisets present at any moment in the existing m regions of the system. Transition between two configurations is performed by using evolution rules also present in each region.*

*Among main Transition P-systems characteristics are massive parallelism and non determinism. This work is part of a very large project and tries to determine the design of a hardware circuit that can improve remarkably the process involved in the evolution of a membrane. Process in biological cells has two different levels of parallelism: the first one, obviously, is the evolution of each cell inside the whole set, and the second one is the application of the rules inside one membrane. This paper presents an evolution of the work done previously and includes an improvement that uses massive parallelism to do transition between two states. To achieve this, the initial set of rules is transformed into a new set that consists in all their possible combinations, and each of them is treated like a new rule (participant antecedents are added to generate a new multiset), converting an unique rule application in a way of parallelism in the means that several rules are applied at the same time. In this paper, we present a circuit that is able to process this kind of rules and to decode the result, taking advantage of all the potential that hardware has to implement P Systems versus previously proposed sequential solutions.*

***Keywords***: *Transition P System, membrane computing, circuit design.*

***ACM Classification Keywords***: *D.1.m Miscellaneous – Natural Computing*

## Introduction

Transition P-systems or Membrane Computing (designed by [Păun, 1998]) are based on the processes that occur among living cells. The idea behind it is the fact that a living cell may change its state depending on the set of elements that are present in it and, of course, depending on the chemical rules that can transform them. Based on this, we can create a computational model based on that behavior. So, there is a definition of a cellular structure that contains elements that can be repeated, conforming multisets, and rules that define how multisets are combined to reach cell evolution. One of these structures (membranes) may contain another ones, conforming a hierarchical relation whose components may communicate among them, always based on what the rules allow. Evolution due to a rule application may cause that a membrane passes information to the one immediately superior in the hierarchy or to any of the ones that are in a level immediately inferior. All this, besides the fact that eventually, a membrane may be inhibited or dissolved by means of some rule application, and that they may have different priorities, does P-systems very interesting in order to define their hardware implementation.

All these processes can be viewed as computational ones and P systems have been sufficiently characterized from a theoretical point of view and their computational power has been settled. However, nowadays, the way in which these models have to be implemented is still a problem not solved. This problem is having two different approaches: software and hardware models. There are many papers about software tools implementing different P system variants [Gutierrez-Naranjo, 2006], but in the case of P-systems hardware implementation, only a few references can be found: connectivity arrays for membrane processors [Arroyo, 2004], multisets and evolution rules representation in membrane processors [Arroyo, 2004b] or a hardware membrane system description using VHDL [Petreska, 2003]. However, in [Martinez, 2006a] and [Martinez, 2006b] there is a hardware approach that implements a circuit that covers the whole process that takes place inside a membrane. Authors describe the way a sequential circuit may control the application of active rules in a Transition P –system and its internal structure.

Being aware that P-systems are defined as "distributed, massively parallel and non deterministic", we think these characteristics should be strengthen. Parallelism takes place in this model in two different levels: the first one is

due to the fact that every cell or membrane evolutions at the same time than the others, and the second one is due to the fact that rules inside each membrane may be applied at the same time.

It is at this point where this work pretends to be positioned: parallelism by means of application of multiple rule at the same time.

The structure of this paper presents, first, the problem and its methodological solution and afterwards, shows its data model and a general representation of the circuit, as well as each part in detail.

## The algorithm

As we may read in [Martinez, 2006a] and [Martinez, 2006b], a hardware approach to P-system is possible. These papers show how the general algorithm of an evolution system may be developed with a circuit. Authors clearly improved the basic algorithm by the way of the proposal of obtaining the number that represents the maximum times each rule could be applied to the current multiset. This number, called *applicability MAX*, is the higher limit for a random number that indicates how many times the rule will be applied, modifying the basic algorithm as:

Let R be the initial set of active rules, $R = \{R_1, R_2, \ldots, R_n\}$ and W the initial multiset, being *input*($R_i$) the antecedents for rule $R_i$

1. R ← InitialActiveRules
2. REPEAT
3. $R_i$ ← Aleatory (R)
4. *MAX* ← Applicability ($R_i$, W)
5. IF *MAX* = 0
6. THEN R ← R - {$R_i$}
7. ELSE
8. K ← Aleatory(1, *MAX*)
9. W ← W – K * input($R_i$)
10. count(K, $R_i$)
11. UNTIL |R| = 0

As we can see, the algorithm works by selecting randomly one rule until there are no rules to apply (|R| = 0). Once the rule is selected, it calculates its *MAX* value; if this value is zero, it means that the rule is no more applicable and it has to be removed from the set of rules.

Afterwards, it generates a random number *K*, equal or less than *MAX* and the application of the rule consists in subtracting *K* times the antecedents input($R_i$) from W. This means that such rule is being *K* times used. Of course we have to store this value so we can check how many times a rule has been applied (step 10).

So, this algorithm is implementing some way of parallelism (in each iteration, a rule is applied K times). However, the importance of parallelism in this kind of model, as well as its possible importance in the field of NP problem solving, urged us to find a way to be able to apply several rules at the same time, improving its throughput (after all, the exposed algorithm just calculates *MAX* for one rule). Thus, the idea is to find a way to select several rules and apply them over the multiset in each evolution step. We could see that this could be achieved in a better way, improving its computational throughput just by considering the initial set of rules as a new set composed by the rules that result form calculating the power set P(R) from the original set of rules. So if we have:

$R = \{R_1, R_2, \ldots, R_n\}$

its power set is:

$P(R) = \{\varnothing, R_1, R_2, \ldots, R_n, R_1 R_2, \ldots, R_1 R_n, \ldots, R_{n-1} R_n, \ldots, R_1 R_2 \ldots R_{n-1} R_n\}$

*As* $\{\varnothing\}$ is an element with no rules and it has no meaning for this work, the power set minus the empty set will be considered:

$P'(R) = P(R) - \{\varnothing\} = \{R_1, R_2, \ldots, R_n, R_1 R_2, \ldots, R_1 R_n, \ldots, R_{n-1} R_n, \ldots, R_1 R_2 \ldots R_{n-1} R_n\}$

If we consider now this set P'(R) as the initial active rules set, what we are doing is to be able to apply several rules at the same time, by the meaning that if a rule $R' \in P'(R)$ / $R' = R_{x \ldots} R_y R_z$ is chosen, a possible evolution may process the antecedents of several rules ($R_{x \ldots} R_y R_z$) at the same time (as many as conform the chosen element). The algorithm, right now would be:

Let R be the initial set of active rules, R = {$R_1$, $R_2$, …, $R_n$ } and W the initial multiset, being *input($R_i$)* the antecedents for rule $R_i$

*Let P(R) be the power set of R and P'(R) = P(R) - {∅} with card( P(R)) = $2^n$ and card(P'(R) = $2^n$ -1) )*

```
1.  REPEAT
2.     ∀ Ri ∈ P'(R), ||      MAXi ← Applicability (Ri, W)
3.     ∀ Ri ∈ P'(R), ||      Ki ← Aleatory(1, MAXi)
4.     COBEGIN
5.        ∀ Ri ∈ P'(R), ||   WT ← Ki * input(Ri)
6.        END ← ¬ ∃ Ki <>0; IF NOT END
7.                        THEN BEGIN
8.                                  Rj ← Aleatory (P'(R)) / Ki <>0
9.                                  COBEGIN
10.                                    W ← W – WT
11.                                    count (Ki, Rj, R)
12.                                 COEND
13.                              END
14.    COEND
15. UNTIL END
```

As we may see, this algorithm underlines the importance of parallelism, taking advantage from the processes that can be done simultaneously. As we will see ahead, there are two types of parallelism: first, some processes are applied to all the rules at the same time (indicated by the sign "||" in steps 2, 3 and 5) and second, some control processes may be done simultaneously (indicated by the clauses "COBEGIN … COEND").

Moreover, differences with the previous algorithm include (steps 2 and 3) calculating applicability *MAX* and a random number ($K_i$, between 1 and its *MAX* value) for each of the rules that are included in P'(R). As they should be calculated simultaneously, process time is not incremented. Once this is done, it calculates the product of each $K_i$ by the antecedents of each rule, but, at the same time this is happening, there is a special process (steps 6 through 8) that selects a random rule but just for the rules whose *MAX* value is different than zero (this means that $K_i$ is also different than zero). This causes that any selected rule is applicable and only in the case that no rule has *MAX* value greater than zero, the *END* condition is reached.

Once the rule is selected, of course the system has to subtract the antecedents ($W_T$) from the set of elements (W) but, again at the same time, it has to decode the participant rules, because not all the rules that are in P'(R) appear also in R. A rule could be the result from the composition of several rules from R and so, the process has to increase the counter for each of the rules from R.

## The model and data representation

Before we can start with the circuit design, there is the need for a definition of a data structure that contains information about the initial membrane state, the initial multiset of objects and the set of evolution rules. Continuing with the work done in precedent papers, and knowing that we have to establish some limits for a suitable circuit, the model should:

a. Limit the cardinality O = {a, b, c, d, e, f, g, h, i, j} of the alphabet to 10.
b. Define the initial multiset involved in a specific membrane *i*, $W_i$, that will be represented by a 4-bits register. The length of this register will be 10. The value in each register position will represent the number of occurrences for the object represented by the alphabet letter in that position.
c. The finite set of evolution rules R associated to the membrane *i* is represented by a set of registers, each of ones represents the antecedents of rule *i*, and the value in each position represents the element occurrences needed for the current rule to be applied.
d. The Application Rules Register is represented by a register which length is, at least, $\log_2 n$, being n = card(P(R))

In this work we have to consider two main aspects:

a. First, the initial set of rules is considered to be the power set of active rules at the beginning of the process. The circuit to obtain active rules from the initial multiset may be obtained from [Martinez, 2006a].
b. Shown solution will be scalable, so, increasing number of initial rules will not have a negative influence in the design (if card(R)=n, then card( P(R) ) = $2^n$ and card(P'(R) = $2^n$ -1) ). In this paper we will work with examples with a set of three initial rules, that makes card (P'(R)) = 7.

The circuit shown in figure 1 takes the set of rules, already P'(R) members (*Initial Active Rules*), and the initial multiset of objects and brings out a complete register (*Application Rules Register*) with the occurrences each rule should be applied to obtain a step of evolution.

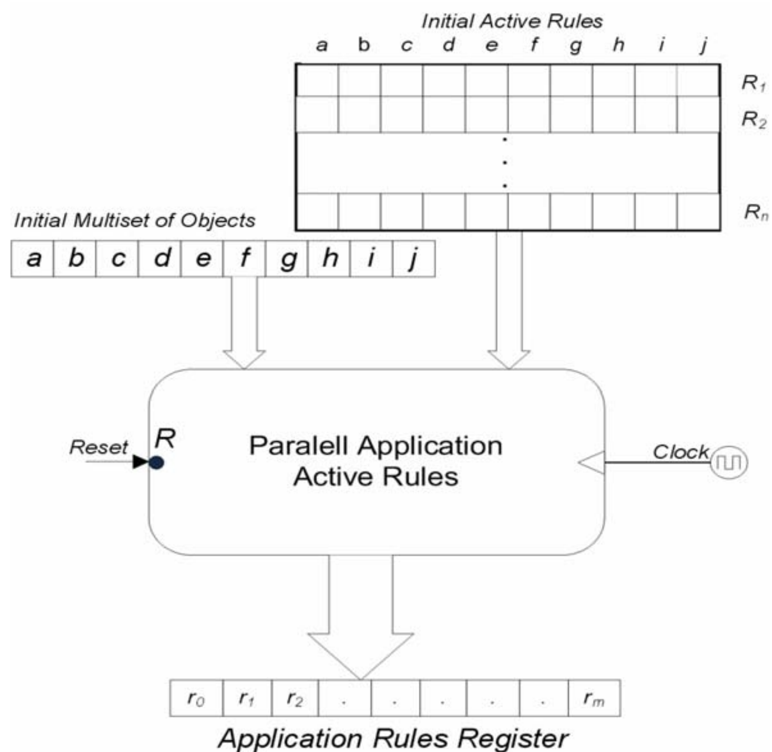


Figure 1. Circuit inputs and outputs

## The circuit

The circuit is the result for assembling different functional units created each one to do a specific job. All of them should be coordinated by a “Logic Control Unit” not represented in figure 2, that takes the control and repeats the whole cycle until the signal provided by the Application Selector F.U. indicates that are no more active rules.

The different units are:

**Applicability MAX F.U.:** This functional unit is the one that receives an active rule and determines its *Aplicability Max* value, as explained before. This value is calculated as the largest number of times current rule can be applied without having in mind the other rules. So, this functional unit needs, as input, the antecedents of current rule and the multiset of objects. The output will be the MAX value for current rule.

The Max value may be obtained [Martinez, 2006a] by dividing each position value from the register for antecedents by its corresponding position value in the multiset register. Once obtained all this results, the smallest one will be the maximum value the rule may be applied.

**Random generator 1..MAX:** once the Applicability Max is obtained, the circuit should generate randomly a value for each of the available rules in the active rules register. This value represents the number that each rule should be applied in case that specific rule is chosen to be the one that consumes the elements and its lowest value will be 1 and the highest will be $Max_i$.

It is very important to realize that Max value could be zero, due to the fact that a rule could not be applied because there are no enough elements in the multiset. In another type of circuit, this would cause the rule to be invalid for the process and that could be a problem. In this case, this is solved by the Application Selector F.U. where a k not equal to zero is selected.

If n is the cardinality of P'(R), all this calculation (n times a random number between 1 and $Max_i$) may be done at the same time, forcing the higher parallelism.

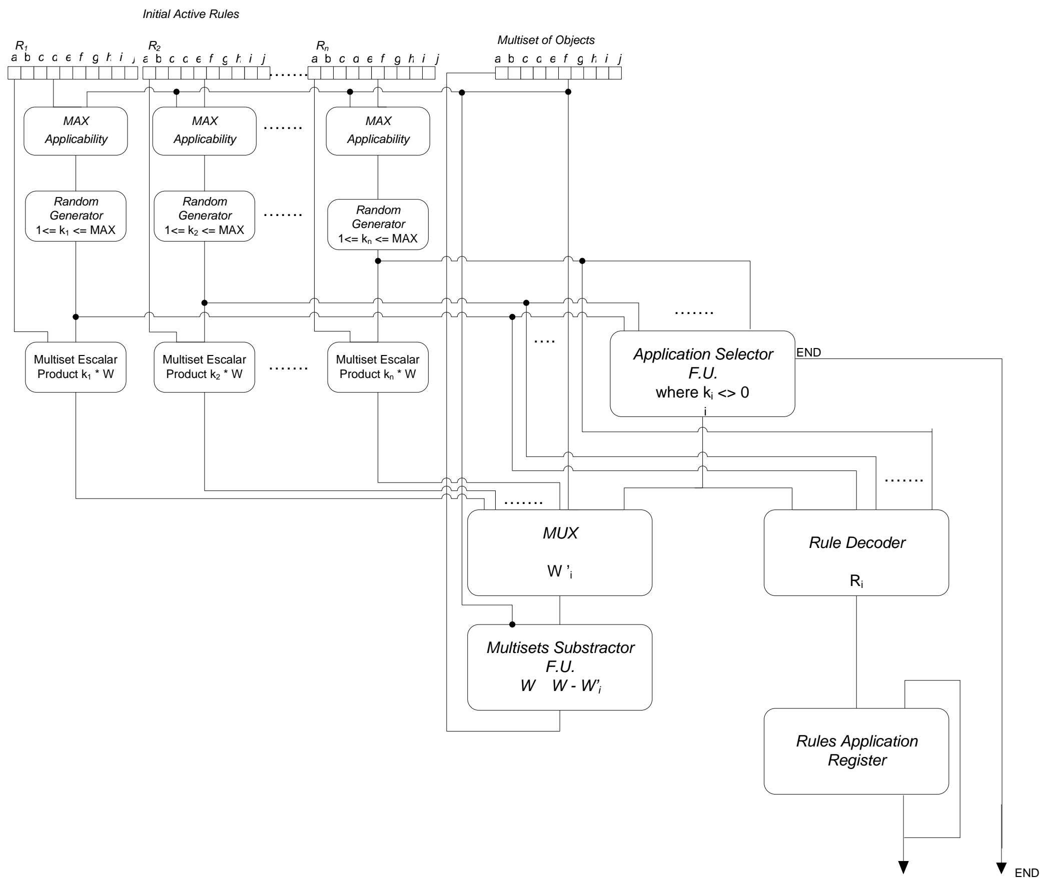


Figure 2: Circuit functional units

**Application Selector F.U.**: Once the previous random generator has calculated $k_i$ for each rule, we can find that any of these numbers may be zero (due to the fact that its *Application Max* value may be equal to zero). We have to implement a way of avoiding to choose a rule with $k_i$ equal to zero because it would cause a delay time in process dedicated, probably, to recalculate a new $k_i$. To avoid this kind of problems, we developed a functional unit that can generate a random number but just for those rules which k is greater than zero.

Achieving the developing of this functional unit included developing of one special cell that obtains the position of the first "1" appearing in the register, and another cell to get the position of the second "1", and another for the third, and so on. We will have as many cells as number of rules in P'(R). As result of this, we will get together all the positions that have a value for $k_i$ different from zero.

As we can see in figure 3, to do this, first we need to transform k values, that can be greater than one, to another values (1 or 0) representing that $k_i$ has a value greater than zero or not. This can be done with a comparator.

Thus, there is a need to have a specific circuit to detect the first “1” in the register, that would be the position of the first rule that has a non zero value. In figure 3 we can see that there is a comparator that sets the position of the value “1” by deactivating the logical gates after it finds the value. Comparison with values 1 to 7 brings us the value of the position for the first rule that has a non zero value for the random number k. There has to be another specific circuit to detect the value for the second rule, the third, etc. Of course, these circuits are similar to the one shown but they ignore the registers behind the position they are looking for.

If we call each of these circuits A, B, C, D…, all of them should be added as we can see in figure 4, in such a way that the first values are all different from zero. Now, all we have to do is to generate a random value no greater than the position of the last number greater than zero. To achieve this, we just have to add the number of values different from zero that are stored in the register and use it as the input for the random generator. The output will be a number between 1 and the number of values different from zero. If we use it as the index for the multiplexer, we will obtain always a value indicating the position of a rule which random k is different from zero and so, we are sure the rule is applicable and active.

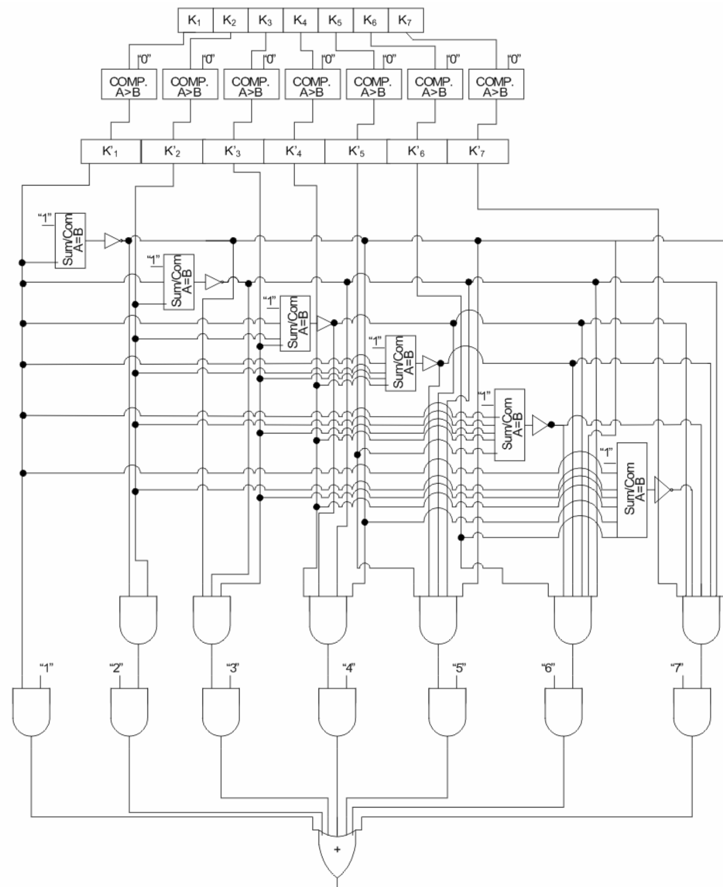


Figure 3: Detecting first rule with $k_i > 0$

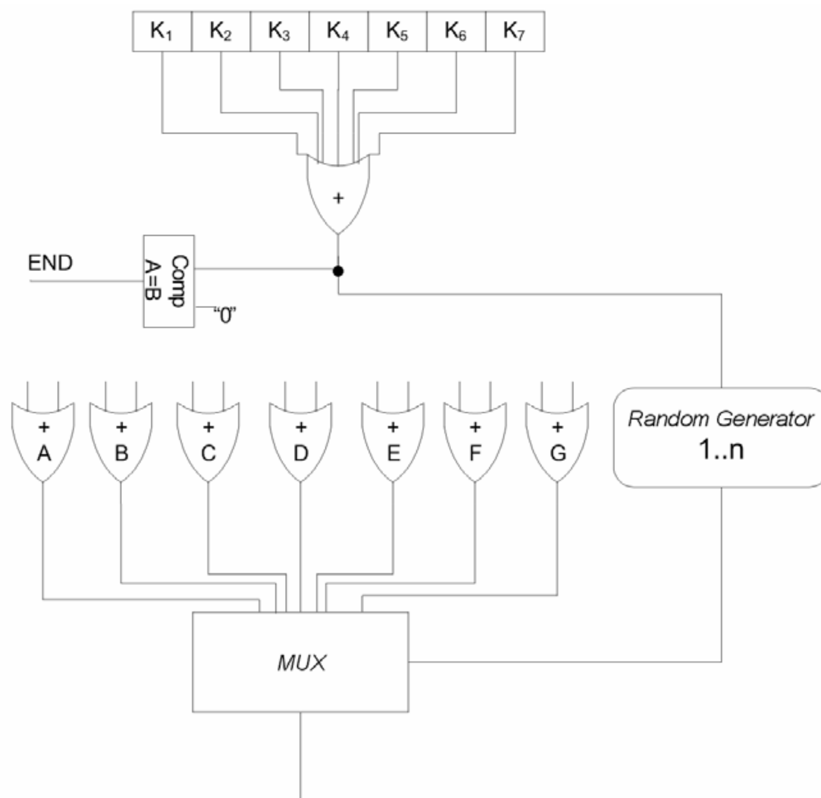


Figure 4: End signal and output for random generator $k_i <> 0$

Of course, if no k value is different from zero, the addition would result in a zero value, which, once compared with “0”, results in the "*END*" signal for all the circuit because it means that no more rules are applicable.

**Rule decoder:** As we can see in figure 1, once a rule is chosen by the *Application Selector F.U.*, we need to perform two different processes: the first one is to calculate the occurrences of elements used to be able to decrement them from the multiset of objects. But there is still another problem: we should be able to register in the *Application Rules Register* the number of times each rule was applied. This means that if the rule applied i was one that belonged to P’(R) but was not in R (possible due to the way we conformed P’(R)), we have to “decode” that rule to the set of rules that conformed R. Circuit in figure 5 shows how it can be done.

The first set of comparators select the rule indicated by the functional unit “*Application Selector*”. As we just have seen, the value for the selected rule, $k_i$, can not be zero. Once we this value, we have to separate the components that conform it.

So, in the example with 3 rules for R ({ R1, R2, R3}) and 7 rules in P’(R):

P’(R) = { $R_1$, $R_2$, $R_3$, $R_1$ $R_2$, $R_1R_3$, $R_2R_3$, $R_1R_2R_3$}

If rule 1 is selected, the circuit will add only the k value for this rule (gate at the left), but if rule 7 is selected, then it will add the k value for rules $R_1$, $R_2$ and $R_3$ because rule 7 is $R_1R_2R_3$ and all of them were applied k times.

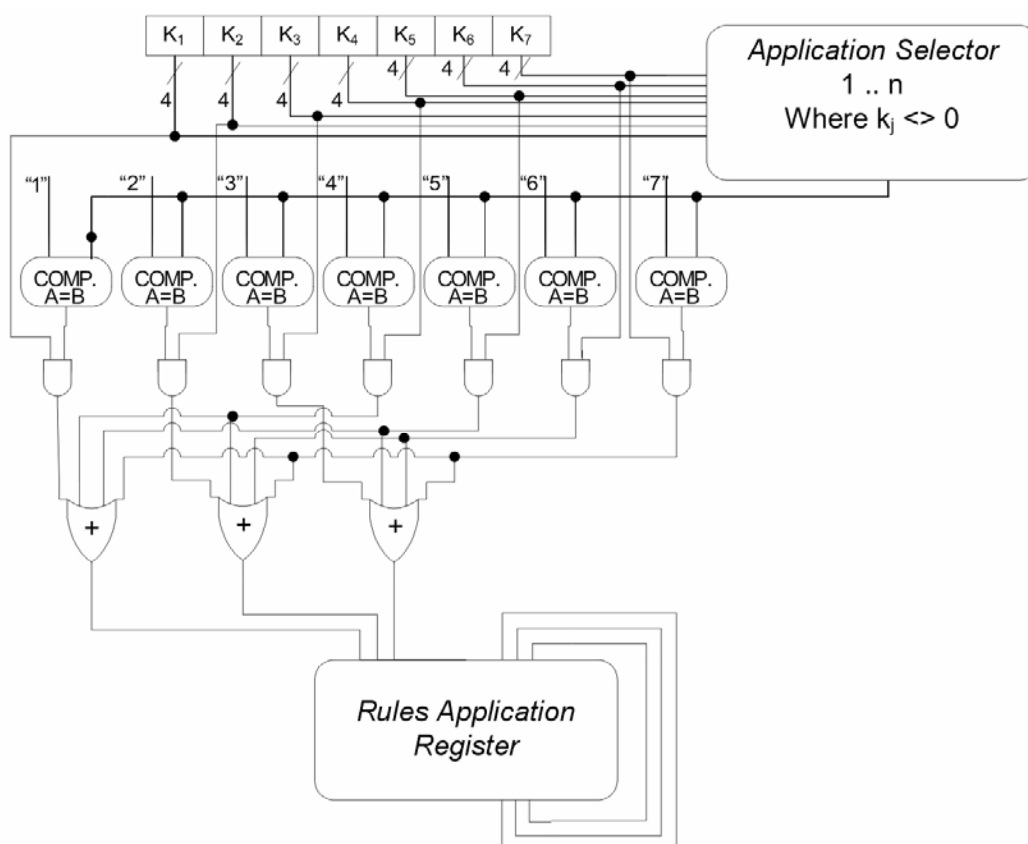


Figure 5: Rule decoder

Whenever the *Application Selector F.U.* enables de *END* signal the "Rules Application Register" will contain the final result, that is, the number of times each rule has to be applied to go forward with a transition. This number is referring the initial set of rules R.

**Other functional units:** Of course, there are more functional units that are in charge of calculating the final amount of elements that the circuit used during each step of evolution. There is a unit that is dedicated to calculate, for each rule, the result of multiplying k (random number generated by the first generator) by each *input($R_i$)* (elements in the antecedent of each rule). Of course this can be done for all the rules at the same time and it just needs a multiplier per rule.

The second one is just a multiplexer in charge of receiving the rule number (j) selected by the *Application Selector F.*U. and to select, according with it, the product $k_j$ * *input($R_j$)*.

Once this is done, we need just to decrement the product selected before from the global multiset, and this is the job for the last functional unit, storing its result in the Multiset Register of Objects to allow a new selection of a rule and let whole process go again.

## Conclusion

Nowadays there are several projects trying to conform different types of circuits to implement membrane computational model with hardware, obtaining active rules and forcing the system to evolution and obtain the number of rules applied. This paper presents how to improve this kind of circuits by emphasizing the massive parallel character P-systems have.

The circuit provides the number of times each rule should be applied to do a complete transition between two configurations, according to its initial set of rules and initial multiset of objects. Of course, different applications over the same sets, do not have to produce the same result.

Hardware implementation is based on basic components like registers, counters, multiplexers, logical gates and so on. The development of the system can be done using hardware-software architectures like VHDL and physical implementation may be accomplished on hardware programmable devices like FPGA's.

## Bibliography

[Arroyo, 2004a] F. Arroyo, C. Luengo, Castellanos, L.F. de Mingo. A binary data structure for membrane processors: Connectivity Arrays. A. Alhazov, C. Martin-Vide, G. Mauri, G. Paun, G.Rozenberg, A. Saloma (eds.): Lecture Notes in Computer Science, 2933, Springer Verlag, 2004, 19-30.

[Arroyo, 2004b] F. Arroyo, C. Luengo, Castellanos, L.F. de Mingo. Representing Multisets and Evolution Rules in Membrane Processors. Pre-proceedings of the Fifth Workshop on Membrana Computing (WMC5). Milano, Italy. June 2004, 126-137.

[Gutiérrez-Naranjo, 2006] M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Nez. Available membrane computing software. In G. Ciobanu, Gh. Paun, M.J. Pérez (eds.) Applications of Membrane Computing.Berlin, Germany. Springer Verlag, 2006. pp.411-436. ISBN: 3-540-25017-4.

[Martínez, 2006a] V. Martinez, L.Fernández, F.Arroyo, I.García, A. Gutierrez. A HW circuit for the application of Active Rules in a Transition P System Region. Proceedings on Fourth International Conference Information Research and Applications (i.TECH-2006). Varna (Bulgary) June, 2006. pp. 147-154. ISBN-10: 954-16-0036-0.

[Martínez, 2006b] V. Martínez, L. Fernández, F. Arroyo, A. Gutierrez. HW Implementation of a Bounded Algorithm for Application of Rules in a Transition P-System. Proceedings on 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC-2006). Timisoara (Romania) septiembre, 2006. pp. 32-38.

[Păun, 1998] Gh.Păun. Computing with membranes. Journal of Computer and System Sciences, 61 (2000), and Turku Center for Computer Science-TUCS Report No 208, 1998.

[Păun, 1999] Gh.Păun. Computing with membranes. An introduction. Bulletin of the EATCS, 67, 139-152, 1999.

[Petreska, 2003] B. Petreska and C. Teuscher. A hardware membrane system. A. Alhazov, C. Martin-Vide, Gh. Paun (eds.):Pre-proceedings of the workshop on Membrane Computing Tarragona, July 17-22 2003, 343-355.

## Authors' Information

***Santiago Alonso Villaverde*** *– Natural Computing Group of Universidad Politécnica de Madrid. - Dpto. Organización y Estructura de la Información de la Escuela Universitaria de Informática, Ctra. de Valencia, km. 7, 28031 Madrid (Spain); e-mail: salonso@eui.upm.es*

***Luis Fernández Muñoz*** *– Natural Computing Group of Universidad Politécnica de Madrid. - Dpto. Lenguajes, Proyectos y Sistemas Informáticos de la Escuela Universitaria de Informática, Ctra. de Valencia, km. 7, 28031 Madrid (Spain); e-mail: setillo@eui.upm.es*

***Fernando Arroyo Montoro*** *– Natural Computing Group of Universidad Politécnica de Madrid. - Dpto. Lenguajes, Proyectos y Sistemas Informáticos de la Escuela Universitaria de Informática, Ctra. de Valencia, km. 7, 28031 Madrid (Spain); e-mail: farroyo@eui.upm.es*

***Javier Gil Rubio*** *– Natural Computing Group of Universidad Politécnica de Madrid. - Dpto. Organización y Estructura de la Información de la Escuela Universitaria de Informática, Ctra. de Valencia, km. 7, 28031 Madrid (Spain); e-mail: jgil@eui.upm.es*