IMPLEMENTING TRANSITION P SYSTEMS

Santiago Alonso, Luis Fernández, Víctor Martínez

Abstract: *Natural computing is a whole area where biological processes are simulated to get their advantages for designing new computation models. Among all the different fields that are being developed, membrane computing and, more specifically, P Systems, try to get the most out of the biological cell characteristics and of the chemical processes that take place inside them to model a new computation system.*

There have been great advances in this field, and there have been developed a lot of works that improved the original one, developing new ideas to get the most of the different algorithms and architectures that could be used for this new model. One of the most difficult areas is the actual implementation of these systems. There are some works that try to implement P Systems by software simulations and there are some more that design systems that implement them by using computer networks or specific hardaware like microcontrollers. All these implementations demonstrate their validity but many of them had the lack of some main characteristics for P Systems.

As continuation for some earlier published works, present work pretends to be the exposition of the design for a complete new hardware circuit that may be used to develop a P System for general purpose, complying with the two main characteristics that we consider more important: a high level of parallelism (which does these systems specially indicated to solve NP problems) and the fact that they should be non deterministic.

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

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

Conference topic*: Membrane Computing*

Introduction

Membrane computing was first introduced by Georghe Păun in 1998 in his article "Computing with Membranes" ([Păun, 1998]). In it, Păun proposes a new computation model based on the structure and behavior of the biological cells existing on nature. He takes advantage of their characteristics to define the possibility of having a structure in which take place multiple processes in a non deterministic way. Each process “fights” for consuming the resources that are present inside that structure and may generate some new elements. These variations along the time may be considered as computation and the final result may be represented by the final stat of the structure.

Membrane computation does not intend the modeling of the cells natural behavior. This would be more appropriated for "bioinformatics" and should have as main goal, the understanding of such natural system for biological investigations. Such as we have just previously exposed, membrane computation does pretend to use the idea of computation by the means of "state changes" that a cell may have, allowing the final state to be considered as the "result".

To use these ideas, a hierarchy for membranes is defined. A membrane will be, not only the "separator" for a specific region, but also its content, that may be composed by:

- A set of different elements, which represent the different chemical materials that are located inside it and that may react among themselves to produce a change in their state.

- A set of rules that define how previously mentioned elements may evolve. These rules should be the equivalent to biochemical rules that govern reactions inside biological cells.
- One or more "inside" membranes that separate regions with the same structure than current.

There are some more cell characteristics but for this presentation we will just consider one more (that will not conform part of our final system). This characteristic is “selective permeability”. It is well known that cell's membrane are not completely impermeable and that they may allow the way in, through themselves, for specific elements that pass from the exterior to inside and vice versa, either by active transportation (with energy consumption) or by passive one (for example, by osmosis).

This capacity to create "pores" is represented in the model by the indication of the target membrane for the result of a rule application (written in parenthesis).

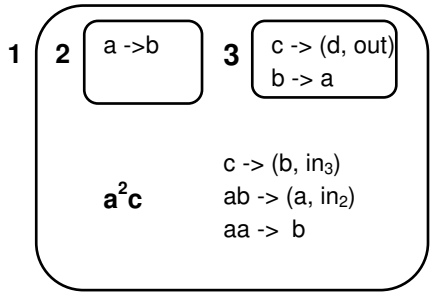


Fig. 1: Simple structure for a membrane

In this figure we represent the fact that a reaction among two elements "a" and "b" from membrane numbered as one would result in the production of an element for class "a" that would be sent to membrane "two", and the existence of an element "c" would result in the generation of an element "b" that would be passed to membrane numbered as three. Inside membrane three we may see that the reaction for an element "c" will result in the generation of an element "d” that should be passed to the membrane just superior in the hierarchy.

This representation is the one considered by so denominated "P Systems” that are, therefore, defined by a membrane structure that contains a multiset of objects, evolution rules that determine how the existing objects in the region change and are transformed, and a series of input/output indications.

Now, it is important to underline two main characteristics for these systems: rules application must be done in a non deterministic manner in the sense that reactions that take place among different elements may have the same priority and in coming scenarios could be executed in a different order. The second important characteristic is the fact that all the rules are applied in parallel, and so, combined with the non-determinism, may result in the execution of any of all the possible evolutions in that scenario. But not only all the rules are applied in parallel, but it is important to realize that all the membrane evolve together, in such a way that if we consider the passing of elements from one membrane to another, this could influence highly in the different states that a membrane could go through. This determines the existence of a “communication” step after the “evolution” step, in which membranes pass their “products” to the another ones indicated by their own rules.

Of course, computation comes from the different "configurations" that the system is going to take, understanding as such each of the resultant states from applying as many times as possible each of the evolution rules over the elements inside the membrane. Final result may be represented by, for example, remainder elements once that all the possible evolutions have been done, or by the elements that the superior membrane puts outside in the last evolution step.

In this work we have to notice that, without forgetting that a membrane may need multiple evolution steps and as many as communication steps with the other membranes, the study is focused on how rules application may be

done in a specific evolution step and the final goal for this document is to obtain a design for a circuit that solves those rules application, without worrying about the communication phase among them.

Some background

Since Păun introduced P Systems, there has been done a lot of work in different approaches: the development of the model, the theoretical study of the different possible solutions or the direct application of this "know-how" through its implementation either using software or hardware. In all these fields, there are a lot of very important references that concern P Systems, as the one in which Păun defines P Systems [Păun, 1998] and, of course, others where the theoretical model is developed. There are, also, a lot of very good works from Fernández ([Fernandez et al., 2006]), Tejedor ([Tejedor et al, 2007]), or Gil ([Gil et al, 2008]) that cover the area which tries to find better algorithms and architectures to be implemented in P Systems. Concerning practical investigation, we may talk about "software simulations", finding very good works as the ones from Ciobanu [Ciobanu & Parashiv, 2001] or the ones from Nepomuceno [Nepomuceno, 2004].

As we may see, there are many more works that we could refer here, but, as far as this work is concerned, we need to focus on the hardware implementations, that would be our background. Solutions for these hardware implementations are approached from different points of view:

First we may consider the use of computer networks, with a good representation by [Syropoulos et al., 2003]. Second, we may see the developing of these systems using microcontrollers ([Gutiérrez et al., 2006]). The last working area about implementations is the one that focuses on the same goal than the ones already related but using FPGAs as processing devices instead of microcontrollers. A FPGA is a device which main characteristic is that it may be configured by the customer using a hardware description language (HDL). Its logic and components allows it to be used in complex circuits. Is in this specific field of Transition P System implementation where we are going to focus from now on.

The first outstanding work was done by Petreska and Teuscher in [Petreska & Teuscher, 2003], in which they designed what they called a “universal component", that allows membrane implementation in specialized hardware such as FPGAs. Martínez in [Martínez et al., 2006] presents the design for a new circuit that finds the set of active rules. Nguyen also presents in an excellent work ([Nguyen et al, 2007] and [Nguyen et al, 2008]) a design for a hardware system (Reconfig-P) implemented using FPGAs that develops all the evolution steps in membranes. Finally, in this specific field of designs using FPGAs, the authors of current work have exposed in [Alonso et al., 2008] a design that works out a circuit that solves rule application inside a membrane. This last work enhances the two already cited characteristics: indeterminism and the highest level of parallelism that may be brought. The result is a design that executes a very high number of operations over the active rules but also the amount of required resources is very high.

Goal

As we said before, the two main characteristics that we want in an implementation for Transition P Systems are:

- The indeterminism with which evolution rules are applied. We will understand as indeterminism the fact that different executions over the same system may not provide the same set of applied rules. If there are rules with the same priority to be applied, any of them will be chosen in a random manner.
- The high level of parallelism that these systems must have. This parallelism occurs at two different levels: the first refers to the fact that all the membranes evolve at the same time. This, which is obvious in "biological world”, is very important for P Systems power. The second level refers to the parallelism

that takes place inside each one of the membranes. In a P System membrane, all evolution rules intend to be applied in parallel and as many times as possible, consuming as many resources as they can.

Once we have emphasized this, we may establish the final goal for current work: to design a circuit using FPGAs that brings a solution for a Transition P System trying to accomplish these two characteristics. However, this circuit won't pretend to cover all the phases nor all the characteristics for these systems. We are going to focus on the evolution phase, that is that one in which the evolution rules are applied until available multiset is empty, without worrying about communication phase. So, we will design a circuit that implements an algorithm that does the active rules application over a given multiset, with a basic characteristic: it must have an universal nature and so, the circuit won't be dependent on initial conditions for the P System and will apply any set of active rules over any multiset (with the limitations imposed by hardware and necessary preloading). On the other hand, the design will simplify the model avoiding characteristics as membrane inhibition or new membrane creation. All the application rules present at a membrane will have the same execution priority.

The method

With the objective of keeping as high as possible the parallelism level during evolution rule application, power set of active rules is used. Power set of a specific set of elements is the one that contains all the possible sets formed by its elements. So, if R is the set of initial rules:

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

Its power set is:

$$P(R) = \{\varnothing, R_1, R_2, \ldots, R_n, R_1R_2, \ldots, R_1R_n, \ldots, R_{n-1}R_n, \ldots, R_1R_2 \ldots R_{n-1}R_n\}$$

As we may see, each of the elements from the power set is the result of one of the possible elements combinations from R. Knowing that $\{\varnothing\}$ is an element that presents no interest for our work, let's consider our power set as the one conformed by all the combinations except the empty one:

$$P'(R) = \{ R_1, R_2, \ldots, R_n, R_1R_2, \ldots, R_1R_n, \ldots, R_{n-1}R_n, \ldots, R_1R_2 \ldots R_{n-1}R_n \} = P(R) - \{\varnothing\}$$

Doing so, considering the power set as the initial set of active rules, what we have is the chance that, at the moment of choosing a rule that is formed by a combination of any others, its application will be the application of all the rules that compose it.

Once we have set which will be the active rules over which we are going to work, we have to say that one of the main problems in algorithms for active rules application is to establish the search space for a solution. There are algorithms that do the application for rules that do not represent a possible solution and so, they have to throw it away to get another one that could be valid. This could be the case for an algorithm that first randomly selects a rule among active ones and after that, checks if there are or not enough elements in the multiset to apply it. Solution for this problem goes through the search for rules only among the space where there are only valid.

One of the possible ways of implementing this vision is to turn around the traditional approach: instead of looking for an active rule and determine if it may be applicable or not, we will proceed to determine, for any possible multiset in our process, which are the rules that may be applicable. This is possible because each superscript for each element in the antecedent shows the requirements for that specific rule to be applied. If we establish intervals with these superscripts for all the elements that are needed in the antecedent and in those intervals we represent the set of rules that may be applied with those elements, what we have are intervals from which what we can get, at each moment, the set of active rules in each iteration. Now, only lasts that from this set of active rules, one of them should be randomly chosen and such rule should be applied a random number of times (between 1 and its maximum applicability factor).

As we are going to use it, we shall remember that we call *"maximum applicability"* for a rule, and we denoted it for "MAX", the maximum number of times that a specific rule may be applied in such a way that it consumes all the resources in the multiset and taking into account that it could not be applied any more due to the fact that wouldn't be enough elements in the multiset to do it.

Let's see the whole proposed algorithm by an example in which we may see the interval creation and the initial preloading for the rules array: let's assume that the initial system has just one membrane, being the multiset the following:

$W = a^3 b^4$

And let the active rules in the membrane be:

$R = \{a^3b \rightarrow c;\ a^2b^2 \rightarrow d\}$

As we already said, with the goal of getting the higher level of parallelism, the first thing that is done is to obtain the power set of R (with the exception of the empty set), that will be constructed by the same rules that are in R, plus the possible combinations among themselves (in our example we will have just one more rule):

$P'(R) = \{r_1, r_2, r_3\} = \{r_1 = a^3b \rightarrow c;\ r_2 = a^2b^2 \rightarrow d;\ r_3 = a^5b^3 \rightarrow cd\}$

Once that the initial set of active rules is established, for each element from all the antecedents in the rules, a set of values is formed and it will represent, in ascendant order, the different superscripts that appear for the element in each of the rules, that, after all, are the needed occurrences for each element for the application of each rule:

$M(a) = \{2, 3, 5\}$

$M(b) = \{1, 2, 3\}$

In our case, there are as many superscripts for "a" than for "b", but it is possible not to be so, because they depend on the number of rules in which they appear and on the possibility for those superscript to be repeated in the rules.

Starting from zero, with each of the superscript we will set intervals that will be closed on their inferior limit and opened in their superior limit, in such a way that each of the superscripts will be exactly the limit. In our example we may write down four intervals for each element, having infinite as superior limit for the last of the intervals:

For element "a":

Superscript	0	2	3	5
Interval	[0,2)	[2,3)	[3,5)	[5, -)

For element "b":

Superscript	0	1	2	3
Interval	[0,1)	[1,2)	[2,3)	[3, -)

These intervals determine "regions" where we may locate the rules depending on how many units of a specific element they need to be applied. So, if a rule needs three "a" elements, it would be in the interval [3,5) for that element and if the same rule (or another) needs just one "b" element, then it would be in the interval [1,2) corresponding to the "b" element.

To complete properly the regions, taking into account that a rule needs the combination of different elements for its application, what we do is to build an array that in each dimension represents one of the existing elements with the intervals shown before, in such a way that any array cell is the region where we can find the rules that may be applied with the specific amount of those elements (indicated by the interval). This may be seen clearly by building the array for our example, where we can see that each element is a dimension (just two elements means two dimensions: rows and columns) with their intervals:

b\a	[0,2)	[2,3)	[3,5)	[5, -)
[0,1)	-	-	-	-
[1,2)	-	-	r_1	r_1
[2,3)	-	r_2	r_1 r_2	r_1 r_2
[3, -)	-	r_2	r_1 r_2	r_1 r_2 r_3

This array is filled up in its cells with the set of active rules that would be applicable if the multiset of elements has as many elements as the intervals for each element ("a" and "b") indicate. Of course, once a rule appears in a cell, it must appear in all the higher intervals in the corresponding row or column for the cell, because those intervals indicate that there are more elements than the needed ones. As well, the cell corresponding to the intersection of higher intervals for each element should contain all the existing rules because it indicates the existence of as many elements occurrences as needed to apply any of the rules.

The array is an "application map" that indicates us the set of rules that may be applied depending on the number of occurrences for each element that are in the multiset. So, if the multiset has only one occurrence of the "a" element, we can see that there would be no applicable rule because in the column under interval [0,2) there are no rules. This is easily verifiable because the three rules we have need three, two and five occurrences of "a" respectively.

If current multiset has, for example, four occurrences for "a" and two occurrences for "b", looking the cell corresponding to the column [3,5] for "a" and the row [2,3) for "b", we may check that applicable rules would be r_1 or r_2.

Knowing that the creation and initialization process for this array may be done before any evolution step that takes place inside the membrane, we may consider it as input data for the circuit we intend to design.

Once the prior phase has concluded, what we get is that, knowing a specific multiset, we have to find the cell that corresponds with the crossing for intervals indicated by the superscripts of each of the elements from that multiset. Once the cell is located, applicable rules are gotten from inside it. After this and with the goal of guarantying the indeterminism, one of them is randomly chosen and will be applicable a random number of times. Besides it, to avoid problems if this number is greater than the times that could be applicable with the elements in the multiset, this number will be limited to the maximum number of times that the rule may be applicable over the multiset. This is equivalent to calculating MAX (maximum applicability) and randomly generate the number inside the interval [1,MAX].

We could represent the algorithm as follows:

Phase 1

1. Power set P' creation for the set of rules R
2. Array loading with the applicable rules in each interval
3. Interval creation for each of the elements in the rules antecedents
4. Let $W = \{w_1, w_2, \ldots w_n\}$ be the initial multiset

Phase 2

5. REPEAT
6. $R \leftarrow r_i \quad \forall r_i \in Array[w_1][w_2]\ldots[w_n]$
7. $r \leftarrow Aleatory(R)$

8. MAX ← Applicability(r)
9. K ← Aleatory(1, MAX)
10. W ← W – {K * input(r)}
11. count (K, P'(R))
12. UNTIL | R | = 0

As we can see in phase two, its operating is similar to others already shown in previous works but with the advantage of knowing that the selected rule is always applicable (step 7). During step 8, its maximum applicability is computed to go on generating the number of times that really will be applied (step 9). The only thing left to do is the subtraction of the spent elements from the multiset (step 10) and register the rule for ulterior queries (we have to notice that we are working with the power set but what we want to know is the set of applied rules from the original set). This process will be iterative until the selected array cell will bring an empty set of rules.

The circuit

Data structure and its representation

Knowing that the goal for this work is a circuit design, we cannot forget that hardware imposes a series of limitations that in theory may not exist. So, for example, the number of elements in a multiset must be limited, and so occurs with the number of rules, etc. To be able to deal with these specifications, we impose the following restrictions and representations:

- Each element from the multiset and each of the elements from the evolution rules will be represented by a set of symbols corresponding to a finite alphabet. Cardinality for this set, just for this work, will be eight, in such way that we will be able to address them with three bits:

  $O = \{a, b, c, d, e, f, g, h\}$
- Multiset will be represented, then, by an array with eight elements
- Correspondence between elements and the alphabet symbols will take place through the position that occupy in the alphabet and in the array, being the same for both.

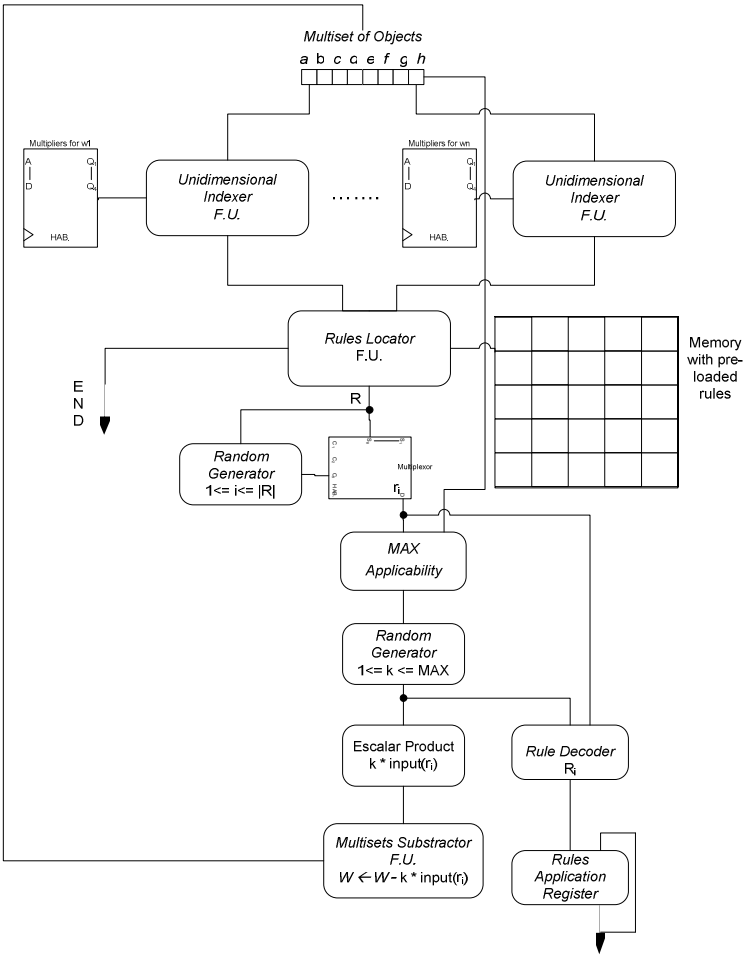
The array, in each position, will contain a number that refers to the amount of occurrences that exist in the multiset for the corresponding element.

The problem associated with the requirement of using a multidimensional array does not require more than a formula that translates any position in such array to another position in a one-dimensional array.

General design for the circuit

The general design of the circuit basically matches with the development, step by step, of the mentioned algorithm, with some adaptations due to the hardware limitations. Of course, all the first phase, including the array preloading must be done by software. As we may see, the available multiset is represented by a register with the number of occurrences for each element. From here, the different functional units that solve each of the steps in the algorithm are represented in the figure.

In a first stage, there are some functional units called "unidimensional indexer" that are in charge of locating, for each of the elements in the multiset, the interval where they belong. To do this, these units have a register with the superscripts already in ascendant order and that have been loaded at the same time that the rules vector. These units will get the indexes for each of the interval to which the superscripts belong.



Once all the indexes are obtained, a second functional unit ("rules locator") is the one that locates, with those indexes, the set of rules that may be applicable inside the preloaded array and, in a next stage, by a random number generator (between 1 and the cardinality of the set of rules), one of them will be chosen to be applied.

Going on with the same algorithm, in the next stage, the maximum applicability for the selected rule is computed and a random number k is generated that will indicate the number of times that the rule will be actually applied. Of course there are just two more things to do: on one hand, to multiply k by the rule antecedents and subtract the result from the multiset, to get the new resultant multiset. We have to underline that data that is preloaded in the array does not change, and when the current multiset changes, the functional units will locate another intervals to which their superscripts belong.

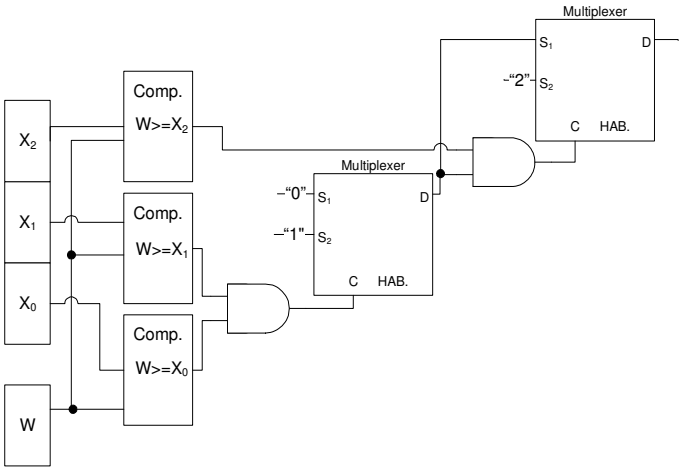
We can see that, along with the multiplication and subtraction of the consumed resources there is a functional unit called "rule decoder" that allows, starting from the applied rule (that belongs to the power set P') to know the rules from the starting set (a transformation from P' to R).

The circuit must keep going on until the functional unit that locates the applicable rules activates the "end signal", that will be when obtained set of rules from the array is empty.

One-dimensional Indexer functional unit

To be able to get the applicable rules set from the multidimensional array we need to know the value for the indexes that indicate the intervals to which the elements from the multiset belong. The goal for this functional unit is precisely to obtain the interval to which belongs the elements superscript. Knowing that one of these units will be in charge of one element, the circuit will have as many units as many elements may have the multiset (its maximum in this work will be eight, as long as we have limited the alphabet to eight symbols).

Input data for the unit are, from the register, the interval limits, while from the multiset, the number that indicates the number of occurrences for the specified element. During first iteration, solution for this module could come from a simple use of comparators, due to the fact that the index value has to be, necessarily, the limit for one of the intervals. Afterwards, during subsequent iterations, the value will be the result of the subtraction of the consumed elements, and this means that the indicated values could be inside the interval, not just in its limits. That's why, to spend a reasonable time, we use a "two by two" elements comparator, that will allow us to decide if the value in the multiset is greater or not than the interval limits.



Unidimensional Indexer for three intervals

In the figure we may see the design for one element from the multiset (W) and just three intervals $[X_0, X_1)$, $[X_1, X_2)$ and $[X_2, \infty)$. Using comparators and multiplexers, the obtained output is the number for the interval to which W belongs, taking into account that the first interval is numbered as zero.

Complexity referring to the number of necessary execution steps for this functional unit is order of the number of existing intervals, that, in the worst case, is a number order of the number of rules in P', that is multiple of $2^{|R|}$.

Naturally, if this functional unit brings the interval number for one element, we will need as many units as elements are, but with the advantage that the process will take place in parallel for all of them.

Memory and rules locator

The proposed design needs less hardware resource than others [Alonso et al., 2008] and, in return, uses a memory module where active rules are stored. This memory must be proportional, on one hand, to the number of elements in the alphabet or number of elements that a rule may contain, because each of them will be represented in one array dimension, that is |W|, and, on the other hand, to the number of intervals that each of those elements has, because in each array dimension we have to have as many columns as intervals has the specified element. Taking into account that the worst case is the one in which a specific element has a different number of occurrences for each of the rules, it seems clear that the maximum number of intervals, for that element, will be equal to the number of rules plus one. The amount of needed memory will be, thus, order of |W|*|R'|, always knowing that the number R' of rules is referred to the power set P' and so, is order of $2^{|R|}$, being |R| the number of rules in the original set. Referring to this original set, complexity will be, then, order of $|W|*2^{|R|}$.

To store the rules that have to appear inside each memory position, we may choose among a binary codification in just a word with the necessary length (one rule is represented in each position) and so, with eight bits we could represent eight rules and it is easy to increment the word's length if necessary. In this codification, a 1 means that the rule is an active one and a zero means "non applicable rule". This system would not increase by itself the amount of needed memory, but would do very much complex the generation of random numbers because they would have to be generated just for the positions that contain a 1 in the word.

Another way of doing this is the selected representation that causes the needed memory to be bigger, because for each of the cells in the array there will be as many words as the maximum number of existent rules and so, we will have all the number of the rules directly from it. So, for example, we will have eight words with three bits each, which will bring us directly the number of the applicable rule. Knowing that the empty element has been deleted from our power set, the zero will mean a non active rule. We have, finally, that the amount of memory is order of $|W|*2^{|R|}*2^{|R|}$, because this is the maximum amount of rules to be stored, or what is the same, order of $|W|*2^{2|R|}$.

The rules locator is the functional unit in charge of locating in the memory the active rules that may be applicable with the current multiset. As input data has the interval numbers to which the element's superscripts belong. With these numbers we may refer each of the active rules array's dimensions. To do so, we have into account that a multidimensional array is just a unidimensional one in which any position may be obtained by doing a simple transformation, multiplying the indexes by the number of elements in each dimension. So, if we are trying to locate a position (i, j, k) in an array with three dimensions which number of elements for each dimension are d_1, d_2, and d_3 respectively, we may do the transformation:

$$Pos(i, j, k) = k*d_1*d_2 + i * d_1 + j$$

What we get is a set of rules that is the input for a multiplexer which output will be selected by a random number generator between 1 and the number of rules obtained in the step just before. With this process the result is a rule that is always applicable.

We have to say that when the rules selector gets an empty set of rules activates the "end" signal and the process is finished.

Maximum applicability functional unit

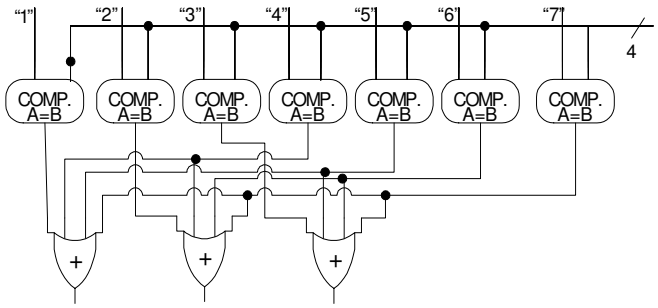
Next step, as we may check in the circuit diagram, is to get the maximum applicability value for the selected rule. To do this, we got the functional unit designed in [Martínez et al., 2006]. Its functioning is based upon the division of the superscript for each element in the rule antecedent by the value indicated in its position in the multiset. Once the results of the divisions are got, the smallest of them will be the one that indicates the number of times the rule may be applied.

Naturally, with the goal of keeping safe the non-determinism, after that, a random number is generated between 1 and that maximum applicability value, and it will indicate how many times the rule will be applied actually.

Other functional units

Once at this point, there is still some work to do and that is why there are some more functional units: the first one gets the result of multiplying the random number by the superscripts of the rule antecedent, and this is the number of consumed elements. Naturally, this number is passed to another functional unit to subtract them from the multiset to be ready for next iteration.

At the same time, the circuit must have a functional unit called "rule decoder" in charge of mapping the applied rule that belongs to the power set to the corresponding ones from the initial set of rules (R). The design for this unit is based upon the one from [Alonso et al., 2008], although this case is simpler because we have already the number of the rule. In the figure we may see the design for a power set (P') of seven rules.



The signals that are activated in this unit are brought to the following unit with the intention of being accumulatively registered and so, to have at the end of all the iterations, the number of times that each rule has been applied.

Conclusion

Until now, the circuit is still in its design phase and although some of the functional units have been tested with simulators, we do not have still real measurements over the circuit and so, we cannot get conclusions upon its real implementation. Besides this, we may establish some guidelines about necessary resources and its behavior.

The design for this circuit pretends to be a balance among the very high cost for hardware resources that the preceding circuit had and the goal for a very high parallelism level during rules application. On the other hand, we may not forget that the circuit represents a universal membrane, in such a way that, after a preloading stage, its use to implement any Transition P System may be possible (naturally, with the limitations imposed for this prototype).

Through the reading of this work, we may easily see that we have achieved a lowering for the number of hardware resources that we needed before and that was exponentially proportional to the number of rules. This enhancement is done, although, with the use of a higher amount of memory associated to the circuit (in which active rules for each interval are stored).

On the other hand, in this circuit a preloading is included and it needs to be executed before the first iteration and this brings some more complexity in terms of processing time. Complexity for this software is order of O (|R|*|W|) because it locates, for each rule, the intervals that has each multiset element.

It also results remarkable the fact that complexity orders in this design goes exponentially high because of the goal to increase parallelism level to the maximum, and so, consider the power set of the rules as the initial set of active rules. In case we want to give up this parallelism increment, we may apply the same circuit to the initial set R and, of course, complexity would be much smaller. In this case, functional unit "rule decoder" would be of no sense because the rule number could be stored directly through a decoder, in the register.

As a final conclusion for this work, we have to underline that the design, in the absence of real tests, seems right towards its implementation as a universal membrane with the exposed characteristics in a FPGA. As we have mentioned, limitations would come from hardware.

Bibliography

[Alonso et al., 2008] S. Alonso, L. Fernández, F. Arroyo, J. Gil. *Main Modules design for a HW Implementation of Massive Parallelism in Transition P-Systems*. Thirteenth Internacional Sysmposium on Artificial Life and Robotics 2008 (AROB 13th). Beppu, Oita, Japan, 2008. Article in "Artifical Life and Robotics" (Springer), Volume 13, pp 107-111, 2008.

[Ciobanu & Parashiv, 2001] G.Ciobanu, D.Paraschiv. *Membrane Software. A P System Simulator*. Workshop on Membrane Computing, Curtea de Arges, Romania, August 2001, Technical Report 17/01 of Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, Spain, 2001, 45-50 and Fundamenta Informaticae 49, 1-3 (2002), 61-66.

[Fernandez et al., 2006] Fernández L., Arroyo F., Tejedor J.A., Castellanos J., *Massively Parallel Algorithm for Evolution Rules Application in Transition P System*. International Workshop (WMC7), Pp: 337-343, Leiden (Netherlands) 2006.

[Gil et al, 2008] Gil F.J., L. Fernández, F. Arroyo, A. Frutos. *Parallel Algorithm for P Systems implementation in Multiprocessors*. International Sysmposium on Artificial Life and Robotics 2008 (AROB 13th). Beppu, Oita, Japan. 2008

[Gutiérrez et al.,2006] A. Gutiérrez, L. Fernández, F. Arroyo, V. Martínez. *Design of a hardware architecture based on microcontrollers for the implementation of membrane systems*, SYNASC 2006, 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing Timisoara, Romania. September 2006.

[Martínez et al., 2006] V. Martinez, L. Fernández, F. Arroyo, I. García, A. Gutierrez. *A Hardware Circuit for the application of active rules in a Transition P System region*. Fourth International Conference Information Research and Applications (i.TECH-2006). ISBN-10: 954-16-0036-0. Varna (Bulgary) Junio, 2006. Pp. 147-154.

[Nepomuceno, 2004] Nepomuceno I., *A Java Simulator for Membrane Computing*. Journal of Universal Computer Science 10, pp. 620-629. 2004

[Nguyen et al, 2007] V. Nguyen, D. Kearney, G. Gioiosa. *Balancing Performance, Flexibility and Scalability in a Parallel Computing Platform for Membrane Computing Applications*. Workshop on Membrane Computing 2007. Thessaloniki, Greece, June 25-28, 2007

[Nguyen et al., 2008] V. Nguyen, D. Kearney, G. Gioiosa. *An algorithm for non deterministic object distribution in P Systems and its implementation in hardware*. Membrane Computing: 9th Workshop, WMC 2008, Edinburgh, UK, July 2008

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

[Petreska & Teuscher, 2003] B.Petreska, C.Teuscher, *A reconfigurable hardware membrane system*. Workshop on Membrane Computing (A.Alhazov, C.Martín-Vide and Gh.Paun, eds) Tarragona, July 17-22 2003, 343-355.

[Syropoulos et al., 2003] A.Syropoulos, E.G.Mamatas, P.C.Allilomes, K.T.Sotiriades. *A distributed simulation of P systems*. Workshop on Membrane Computing (A.Alhazov, C.Martin-Vide and Gh.Paun, eds); Tarragona 2003, 455-460.

[Tejedor et al, 2007] J.A. Tejedor, L.Fernández, Arroyo, G. Bravo. *An Architecture for Attacking the Communication Bottleneck in P Systems*. International Sysmposium on Artificial Life and Robotics 2007 (AROB 12th). Beppu, Oita, Japan. 2007

Authors' Information

Santiago Alonso *– 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 *– 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*

Víctor Martínez *– Natural Computing Group of Universidad Politécnica de Madrid. - Dpto. Arquitectura y Tecnología de Computadores de la Escuela Universitaria de Informática, Ctra. de Valencia, km. 7, 28031 Madrid (Spain); e-mail: victormh@eui.upm.es*