Main Modules Design for a HW Implementation of Massive Parallelism in Transition P-Systems

Santiago Alonso, Luis Fernández, Fernando Arroyo, Javier Gil
Natural Computing Group - Universidad Politécnica de Madrid – Spain
(Tel : +34-91-336-5082; Fax : +34-91-336-7520)
{salonso, setillo, farroyo, jgil}@eui.upm.es

Abstract: P systems are intended to be an alternative computing model for traditional systems. They are designed to emulate the behaviour of living cell systems and as we know, processes inside a cell take place in a "parallel" manner: chemical reactions are taking place all the time and all together. Having this into account, any implementation for such a system (P system), should have as a main characteristic this "parallel" way to do processes. The aim of this paper is to continue a previous work where we got a logical design for a hardware circuit that tries to implement a system wich can be used to emulate the behaviour of a "membrane": it takes a set of rules, selects one of them and applies it according with a specific algorithm. But all of this is done with the spirit of having as much parallelism as possible. Here we present some improvements to the original circuit, presenting an improved module to select valid rules and the detailed design for other specific components.

Keywords: P System, membrane computing, circuit design.

I. INTRODUCTION

As investigation goes on Computer Science, other computing models than the traditional ones appear on the scene. In this sense, Natural Computing studies computational models that are inspired by the dynamic processes that take place on living organisms. An important example are Transition P-Systems (Membrane Computing) which were introduced by Paun [1] in 1998 as a model based on processes that take place inside living cells. It is based on the fact that a living organism, a cell, has the ability to change its state as reactions take place inside it. These reactions are due to the presence of chemical elements that cause the generation of new elements as they interact with each other. All these changes induce a different final state of the cell (if we may consider a "final" state on a living organism). Of course, changes in one cell may provoke changes on its neighbours and may also affect some more cells that are "related" to the original one in different ways.

Based on all this, Paun introduced the Transition P-Systems as systems where an element called "membrane" is a region that may change its state (and so, may be used as a computational element) due to the presence of specific elements that may be repeated, conforming multisets and the existence of rules (*evolution rules*) that define how the elements from multisets may be combined to obtain new ones. All this, like in living cells, may affect other membranes, because they group in a hierarchical way, where a membrane may contain others, defining a relation among them that may be used to pass information ("elements") from one to another that is just underneath or overlies it.

Besides this, eventually, a membrane may be inhibited or dissolved by means of some rule application, and they may have different priorities. Of course, all these properties present a system that appears to be very interesting as a new computational model.

Nowadays, P systems power for computation is settled and there are many approaches to the problems that appear when trying to develop systems that represent them. Once more, we may find two main lines of work: the one that addresses a software solution and the one that focuses on hardware implementations. About the first ones, there are many works referring P Systems variants [2] but there are few that focus on hardware implementations: we may read about connectivity arrays for membrane processors [3], multisets and evolution rules representation in membrane processors [4] or a hardware membrane system description using VHDL [5].

Among the works covering hardware approaches there are several, [6] and [7], that present an architecture based on microcontrollers. These proposals maintain the advantage of software/hardware combination to obtain feasible implementations.

Coming back to the "living world" comparison, we may see that living cells do not present "sequential"

processes. All the chemical reactions take place at the same time and they do not wait one for each other. In fact, this is one of the most important characteristic: a very high level of parallelism. Moreover, we may say that parallelism at living organisms take place at two levels, one is the already meant for the chemical reactions inside a cell, but there is another: the fact that all the cells evolve all together. So, it is important to realize that parallelism should be the main idea when hardware implementations for transition P Systems are designed. This leads us up to the idea to use specific hardware solutions that may increase parallelism and so, and should increase also the throughput of the system.

In [8] and [9] we may read about 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.

But again, we think that parallelism level must be strengthened. Leaving aside the whole set of membranes and focusing on the processes inside just one, this paper pretends to go further on the work done to design a circuit that is able to process more evolution rules at the same time that any other before, just by implementing, with specific hardware, the algorithm that uses the power set of a given set of rules to deal with several rules at the same time [10]. In this work we present enhanced modules for the circuit and some VHDL code for it.

II. PREVIOUS WORK

Taken into account the work done in [8] and [9], we may see that a hardware implementation is possible for Transition P-Systems. Authors improved the basic algorithm for an evolution step calculating the "*MAX applicability*" value that represents the maximum times a rule may be applied according to the number of elements that are present in its region. Thus, a rule selected randomly from the whole set of rules is applied as many times as a random number (generated in the interval from 1 to its *Max* value) denotes. This way of doing it guarantees the nondeterministic character of the circuit and the parallelism by the meanings that the selected rule is applied as many times as this random value. The algorithm eliminates rules when its *Max* value is zero (no longer applicable). It goes on until no rule is applicable.

As we already mentioned, a high level of parallelism should be present in the design of this kind of circuits, and so, we worked on an improvement that could illustrate it. Initial work [10] included the idea of applying several rules at the same time over the multisets in each evolution step. We could achieve this by using the power set P(R) of rules, generated from the initial active rules set. So if we have R as the initial set of active rules:

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

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:

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

Taking this P'(R) set as the initial set for active rules, a possible evolution step may be considering the rule conformed by several basic rules (the ones from R). This will increment the parallelism because we are attending the applying of more than one rule each time an evolution step is done. Of course, the level of parallelism also takes into account that the *Max* value for each rule will be calculated and like in the previous algorithm, rules may be applied more than once (and at the end, this means that more than one rule would be applied more than once in each evolution step). The proposed algorithm is:

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. Let $P(R)$ be the power set of R and $P'(R) = P(R) - \{\varnothing\}$

```
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;
7.           IF NOT END 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
```

This algorithm takes as input the initial active rules set that is, in this case, the power set of the original active rules set and calculates the "*Max applicability*" value for each rule (all of them calculated at the same time, in a parallel manner). Once this is done, it generates a random number for each of the rules (again all at the same time) that indicates the number of times the selected rule will be applied (obviously, this number can not be greater than the corresponding Max value because the rule could not be applied that number of times).

Afterwards, we can see that there are two processes than can be done simultaneously: we take advantage from the fact that the process of selecting a valid rule will take some time to do, at the same time, the calculating for the elements that would be used from the multiset if a specific rule would be selected. Doing so, we do not have to wait for the selector to do its job and we will have ready the result for any of the possible consumptions.

On the other hand, selecting a rule is not easy because it is possible that some of them have a Max value equal to zero (we have to consider that the whole process will have several iterations and so, a Max value may be zero due to its total resources consumption).

Of course, at the end, the number of each used element shall be subtracted from the multiset and the number of times a rule is applied, will be registered, taking into account that this number may have to be decoded, because we want the number of rules used from the initial set R and not just from P'(R).

III. LIMITS

Due to the limitations of any specific hardware implementation, the circuit has the same limitations than in precedent works:

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 one represents the antecedents for rule *i*, and the value in each position represents the element occurrences needed for the current rule to be applied.

a. First, the initial set of rules is considered to be the power set of active rules at the beginning of the process.

Also, in this work we have to consider two aspects: first, the initial set of rules is considered to be the power set of active rules and second, we will limit the circuit in this work to a power set composed by seven rules (that comes from a set with three active rules, *card (R) = 3, card (P(R)) = 8, card (P'(R)) = 7*).

IV. THE CIRCUIT

The circuit tries to represent the implementation for the algorithm as close to it as possible. As we may see in figure 1, it has different phases where each process is done. Up, the initial layer represents the register where initial active rules are stored and the register for the initial multiset of objects.

In a first step, a specific functional unit is the one that will calculate each of the Max values for each of the rules. This value will be the largest number of times a rule may be applied without having in mind the other rules. We may see the design for it [9] and it takes as inputs the multiset and the antecedents for each rule, dividing each position value from the register for the antecedents by its corresponding value in the multiset register. The smallest from all of the results will be the *Max* value for that rule.

In a second stage, another functional unit will generate the random number between 1 and *Max* value. As before, there is one functional unit per rule, so all of them may be generated at the same time. This will be the number a rule will be applied in case that rule is selected by the "Application Selector". It is important to realize that this random number may be zero, due to the rules that, in following iterations have Max value equal to zero.

Taking it into account, we may address the design of the most important functional unit: the Application Selector. This functional unit is so important because it is the one that selects the rule to be applied, but it has to discriminate the rules that have its random value equal to zero because it means that these rules may not be applied.

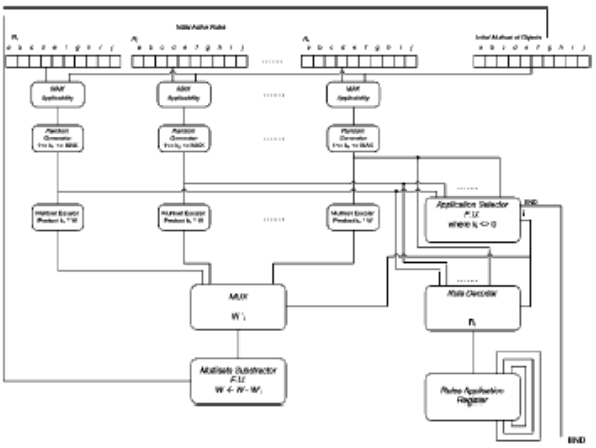

Fig.1. Functional Units

The design done in previous works resulted in a working functional unit but with a problem with the execution time: it was order of 2^n, being n the number of rules. Of course, this time would compromise the higher level of parallelism because the evolution time would have been severely retarded. A new design for it was done and what we have is a functional unit that reduces its execution time to n, where n is again the number of rules.

Thus, the "Application Selector" still has as main function to select a rule which random number is different from zero. This time, the development of this unit is based on a selection for each two rules and it takes into account that if value for rule i, k_i is zero, it has to select the other rule. In case both numbers are different from zero, then a multiplexer is activated and any of its outputs is randomly selected. In case both rules have their *Max* value equal to zero, then zero will be the output.

As we may see in figure 2, first, we do use a "or gate" do determine if content of k_i is zero and we call both outputs ORK_1 and ORK_2. After that, fornulae to obtain the circuit are:

ORK1	ORK2	C1	C2
0	0	0	0
0	1	1	1
1	0	1	0
1	1	1	Random

$C1 = ORK_1 + ORK_2$

if $ORK_1 * ORK_2 = 0$ => C2 = ORK2

if $ORK_1 * ORK_2 = 1$ => C2 = Random

C2 will be selected with a second multiplexer depending on the result of the logic product ORK_1 times ORK_2.

Of course, this works for two rules, but the whole selector will be the concatenation of several modules like this one (as many as needed in each case). Indeterminism is guaranteed with the random selection and, on the other hand, no rule with *Max* value equal to zero will be selected. Whenever a zero is going to be the final output for the whole "Application Selector" functional unit, then the *END* signal for the whole circuit is enabled, because it means that no rule will be applicable.

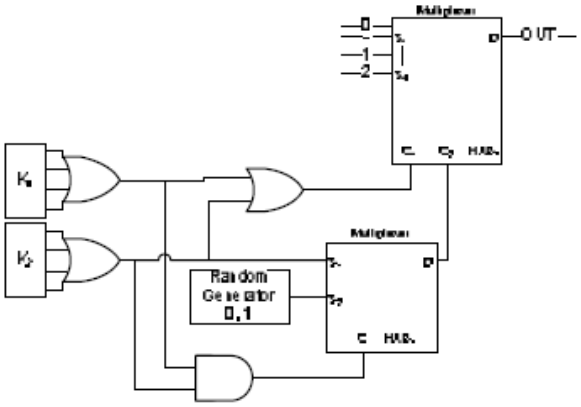

Fig.2. Application Selector Module

At the same time that the rule is being selected, other functional unit is doing the scalar product that brings the result of multiplying, for each rule, its random number k_i times the antecedents it needs to be applied one time and so, obtaining the total amount of objects that would be used in case any of the rules would be selected. Of course, this is done in parallel with the job from the application selector.

Coming back to figure 1, after "level three", there are the two actions the circuit must do: first, receive from the application selector the number of the selected rule and, with a multiplexer, select the corresponding product $k_i * W_i$ and pass it to a "Subtractor F.U." that will decrease the corresponding values in the initial multiset register.

But at the same time this is happening, and once we already know the rule that has been selected, it is received by the "Rule decoder". This functional unit will take the selected rule and it will extract from it the basic rules that are the ones corresponding to the original set R.

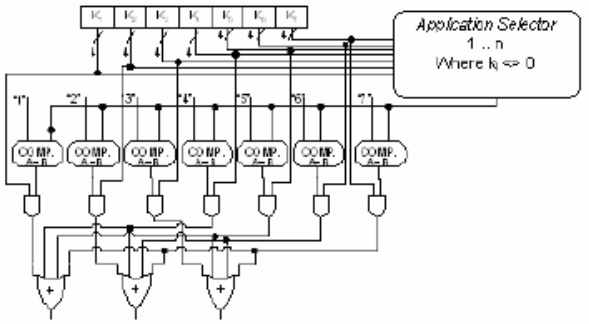

Fig.3. Circuit for Rule Decoder

The "Rule decoder" functional unit has been defined with specific software and it is being tested to ensure its proper operation (fig. 4).

The output from the rule decoder is stored in the "Rules Application Register", which stores the number that is applied each rule, referring to the original set R. Once the circuit ends all its iterations, this register will contain the whole information for an evolution step.

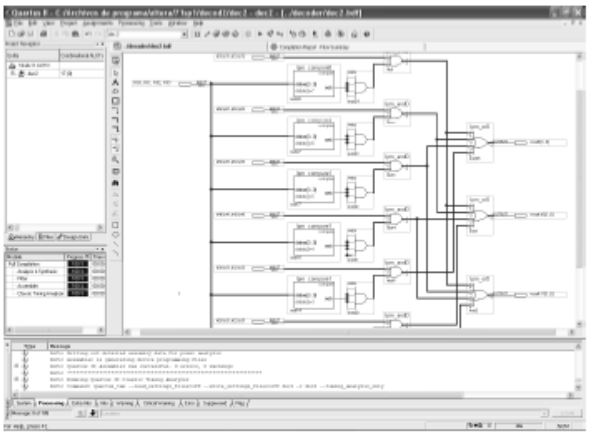

Fig.4. Testing decoder

V. CONCLUSION

In this paper we present the logical design for a circuit that would implement a Transition P System. Furthermore, what we pretend with it is to emphasize parallelism instead of attending to other kind of criteria. Original design was improved by the meanings of a better functional unit for the "Application Selector" that brings a much better processing time and a more detailed approach to the whole solution was shown.

Of course, there is still work to do, but with this kind of circuit we want to show that other feasible approaches to hardware implementations are possible for Transition P Systems. Future work covers writing VHDL for all the functional units and testing for all modules using software tools. The final stage would be using programmable devices like FPGAs for physical implementation.

REFERENCES

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

[2] 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

[3] F. Arroyo, C. Luengo, Castellanos et al. 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.

[4] F. Arroyo, C. Luengo, Castellanos et al. Representing Multisets and Evolution Rules in Membrane Processors. Fifth Workshop on Membrana Computing (WMC5). Milano, Italy. June 2004, 126-137.

[5] 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.

[6] A. Gutiérrez, L. Fernández, F. Arroyo et al. 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 26-29, 2006.

[7] Gutiérrez Rodríguez, A.; Fernández Muñoz, L.; Arroyo Montoro, F. et al. Hardware and Software Architecture for Implementing Membrane Systems: A case of study to Transition P Systems. 13th International Meeting on DNA Computing. June 4-8 2007 Memphis, Tennessee, USA

[8] V. Martinez, L.Fernández, F.Arroyo et al. 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.

[9] V. Martínez, L. Fernández, F. Arroyo et al. 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) september, 2006. pp. 32-38

[10] Santiago Alonso, Luis Fernández, Fernando Arroyo et al. A Circuit Implementing Massive Parallelism in Transition P System. International Conference "Information Research and Applications". 25-30 June 2007. Varna, Bulgaria