Computing and Informatics, Vol. VOL, 2012, 1050–1079, V 2014-Oct-21

FAST HARDWARE IMPLEMENTATIONS OF STATIC P SYSTEMS

Juan Quiros

ID2 Group, Department of Electronic Technology, University of Seville,
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
e-mail: `jquiros@dte.us.es`

Sergey Verlan

LACL, Département Informatique, Université Paris Est,
61, av. Général de Gaulle, 94010 Créteil, France
e-mail: `verlan@univ-paris12.fr`

Julian Viejo, Alejandro Millan, Manuel J. Bellido

ID2 Group, Department of Electronic Technology, University of Seville,
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
e-mail: `julian@dte.us.es,amillan@us.es,bellido@dte.us.es`

Abstract. In this article we present a simulator of non-deterministic static P systems using Field Programmable Gate Array (FPGA) technology. Its major feature is a high performance, achieving a constant processing time for each transition. Our approach is based on representing all possible applications as words of some regular context-free language. Then, using formal power series it is possible to obtain the number of possibilities and select one of them following a uniform distribution, in a fair and non-deterministic way. According to these ideas, we yield an implementation whose results show an important speed-up, with a strong independence from the size of the P system.

MSC 2010: 94-04, 68-04, 68U20

Keywords: reconfigurable hardware, P systems, static P systems, FPGA, membrane computing, parallel implementations of membrane computing, simulator of membrane computing, hardware implementations of membrane computing, parallel implementations of static P systems, simulator of static P systems.

1 INTRODUCTION

The beginning of the membrane computing goes back to the end of the nineties, when G. Păun introduced it [1]. This area gets its essence of living cells and, in consequence, the first models are defined as an hierarchical structure or a tree topology of compartments (membranes that delimits cells), which contains objects (chemicals), which evolves according to applicability rules (chemical reactions). Using this concept as a starting point, more variations, based on more biological processes which take place in living cells, have been detailed into following years (and it continues nowadays). One of the most relevant modifications was the introduction of the tissue P systems [2], in which the tree topology of the structure of membranes is substituted for a graph topology, where the structure is composed by cells, instead of tree nodes [3].

This work is based on the formal framework introduced in [4], in which a general class of multiset rewriting system, which contains P systems and tissue P systems, is designed. So, this section only describes, in a non-formal way, the most relevant aspects for the introduced work. For more details about P systems, we refer to the books [3, 5], and to [4] for specific information relative to the formal framework.

A P system, Π, can be describe as a network of cells

$$\Pi = (n, V, w, Inf, R), \tag{1}$$

where:

n is the number of cells.

V is the alphabet, which contains symbols for each object in the system.

w is a tuple $(w_1, \dots, w_n)$, where w_i corresponds to the multiset of objects ($o \in V$) contained in cell i.

Inf represents which symbols are provided by the environment. In this case, a cell can receive an infinite number of objects. $Inf = (Inf_1, \dots, Inf_n)$, where $Inf_i \subseteq V$ determines which elements of the alphabet are received by cell i from the environment.

R is a finite set of rules, r_i, with the form $r_i : X \to Y; P, Q$. $X \to Y$ are vectors of multisets, which can be written as $((x_1, 1), \dots, (x_n, n)) \to ((y_1, 1), \dots, (y_n, n))$, meaning that objects in X, also called $lhs(r_i)$, left-hand-side of r_i, are consumed by the evolving rule, whereas objects in Y, also called $rhs(r_i)$, right-hand-side of r_i, are produced. Attending to P and Q, they are also vectors of multisets, called permitting and forbidding conditions, but in the form $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_n)$. These last ones affect the applicability of the rule in the following way: a rule can be applied, or is applicable, if all elements in P are contained by the cells and none of the elements in Q are contained by them. Obviously, it is necessary that also exist all elements which are going to be consumed by the rule, *i.e.* all elements in X.

For the design of the simulator, we have only considered static tissue-like P systems. According to above definition, a P system is static when its cells do not change along the computation. Moreover, in order to reduce complexity, we have removed from the original rule definition the permitting and forbidding conditions. As it is shown in section 5, these elements can be added easily to the simulator.

The computation of any P system starts with its initial configuration C_0. A configuration describes the state of the system at a particular time. So, in the case of a static P system, the objects contained in each cell of the system are only required, because of the structure does not change along the computation. In consequence, the computation begins with $C_0 = w$ and, from this state, the system evolves to next configuration C_1 by means of application of rules, what is called a computation step, and continues until a halt condition is reached. We will not enter into details about halting conditions, they are discussed in the bibliography suggested above. However, the most typical conditions take place when the system reaches a configuration in which it is not possible to apply any rule, called total halting condition; or it is equal to previous configuration, *i.e.* although rules which can be applied exist, their application does not change the configuration, called adult halting condition.

For our purpose, the most interesting aspect of P systems is the computation step, *i.e.* how they evolve from a configuration, C_i, to the next one, C_{i+1}. This process is described below.

1. At the beginning of the step, the system has a configuration C_i.
2. Given C_i, there will be rules which can be applied, called as applicable rules, and others which can not. According to some restrictions, called as derivation mode (δ), it is possible to define multisets of applicable rules. We define $Appl(\Pi, C_i, \delta)$ as the set of all multisets of applicable rules which verify the restrictions imposed by δ. We note that *Appl* is associated to a P system Π, a configuration C_i and a derivation mode δ. Hence, computing $Appl(\Pi, C_i, \delta)$ will be the first mini-step. As the reader can deduce, derivation modes play an important role in the semantic of the model. They can be seen as some restrictions which should be accomplished by a multiset of applicable rules, R, to be included in the *Appl* set. There are several types, and we refer again to bibliography for more details. Maximal parallelism, *max*, was the first derivation mode and the most used until now. In an informal way, it says that a multiset of applicable rules, R, can be included in *Appl* if, and only if, it does not exist another rule, not included in R, and which can be applied; in other words, if we remove from the system all objects consumed by rules in R, there will not be enough resources so that

another rule can be applied. Formally, maximal parallelism is defined as:

$$\begin{aligned} Appl(\Pi, C, max) = \{ & R' \mid R' \in Appl(\Pi, C) \text{ and} \\ & \nexists R'' \in Appl(\Pi, C) \text{ such that } R'' \supsetneqq R' \} \end{aligned} \tag{2}$$

3. Once the $Appl(\Pi, C, \delta)$ has been computed, an element R must be selected from it. The criteria which must be followed depends on the type of P system. An example is extracting an element following an equi-probable distribution, *i.e.* all elements have the same probability of being chosen.
4. Finally, the last mini-step is applying the selected R. It consists in, for each rule $r : (X \to Y; P, Q)$ in R, removing all objects contained in X and adding all those ones contained in Y. After that, a new next configuration, C_{i+1}, is obtained.

In this paper we detail basic ideas about simulation of non-deterministic P systems, which choice the applicable set of rules following a uniform distribution. Our point of view is slightly different from other approaches, achieving a performance close to ideal. Although it implies a loss of flexibility, reducing the range of input P systems, it is worth to note that the acceptance of a P system does not depend on its class, as it happens in other simulators, but on the complexity of the dependencies of its rules, achieving a wide range of target P systems. In order to exemplify our approach, we present a hardware implementation using FPGA technology and based on our ideas, with a performance of around 2×10^7 computational steps per second, independently of the number of used rules or types of objects.

This paper is organized as follows. Firstly, in Section 2 we describe previous implementations of P systems simulators, focusing on those ones using FPGA technology. Besides, we give a brief introduction to the theory of formal power series (Subsection 2.1), giving examples of the computation of generating series for different languages. In Section 3 we explain our method of pre-computation of all possible applications of rules (Subsection 3.1), and the general architecture of our simulator (Subsection 3.2). Section 4 gives an example of a FPGA implementation of a concrete P system using our ideas: firstly, we present the mathematical details concerning the example, secondly, subsection 4.1 overviews the specific hardware design for the simulator and lastly, subsection 4.2 presents the obtained results. Finally, in Section 5 we discuss about some improvements and future research tasks, and in Section 6 we summarize our conclusions.

2 PRELIMINARIES

The problem of computer simulation of different variants of P systems arose at the early beginning of the development of the area. The first software simulators [6, 7] were quite inefficient, but they provided an important understanding of the related problems. Since most variants of P systems are by definition inherently parallel and non-deterministic, it is natural to use distributed or parallel architectures in order to achieve better performances [8, 9, 10].

Another fruitful idea is to use specialized hardware for the simulation. This approach was realized in [11, 12] using FPGA reconfigurable hardware technology. The first implementation from [11] is based on region processors which have rules as instructions and multiplicity of objects as data. Although it has several limitations which limit its performance (parallelism across membranes only and a reduced extensibility and scalability), it demonstrates that P systems can be executed on FPGAs. In the other [12, 13, 14] two possible designs are detailed: rule-oriented and region-oriented systems. In the first one, each rule is considered as a basic processing unit and, in consequence, has a specific hardware core. As a result, the system achieves a maximum degree of parallelism, because of all rules are executed in parallel by specific hardware components. In the second case, the basic processing units are regions. Thus, communications between regions acquire more relevance: local rules are processed by the region processors and, after that, a communication process between regions takes place in order to update the multiplicity of objects. In both architectures, there is a control logic which synchronizes the operations of processing units and updating of registers which save the configuration of the system. How registers are grouped and what is considered as a basic processing unit depend on the approach (rules or regions).

An important point for a (parallel) computing platform for membrane computing is to achieve a good balance between performance, flexibility and scalability. This is especially important for hardware simulators because the high performance comes often at an important price of flexibility or scalability. The important drawback of FPGA simulators from [11, 12] is that they suppose that the evolution of a P system is deterministic, and thus these simulators will yield always the same result for the same initial configuration. However, the non-determinism in P systems plays an important role and its absence reduces drastically the classes of P systems that can be used with the above simulators.

Furthermore, these simulators are based on the model detailed in the foundational article [1]. As a consequence, they have strong restrictions about topology, type of objects and rules of the P system. So, a more flexible simulator would be desirable. In that way, some theoretical results, as [4], can be a starting point in order to develop a simulator which covers most of the restrictions about topology, rules and objects, obtaining a more flexible one.

2.1 Context-free grammars as generators of formal power series

We assume that the reader is familiar with some notions from the formal power series theory, especially related to the theory of formal languages. We suggest the reading of [15] for more details on this topic. We denote by $|w|$ the length of the word w or the cardinality of the multiset or set w. For our purposes we consider that a formal power series f is a mapping $f : A^* \to \mathbb{N}$, where A is an alphabet and $\mathbb{N}$ is the set of non-negative integers (in the general case a formal power series is a mapping from a free monoid to a semiring). This mapping is usually written as

$$f = \sum_{w \in A^*} f(w)w. \tag{3}$$

It is known that a context-free grammar $G = (N, T, S, P)$ can be seen as a set of equations $x_i = \alpha_1 + \cdots + \alpha_{n_i}$, for each non-terminal x_i of G, where α_j are the right-hand sides of productions $x_i \to \alpha_j$, $1 \leq j \leq n_i$. A solution of G is a set of formal power series $s_1, \ldots, s_k$, such that the substitution of x_i by s_i in above equations converts them to the identity, *i.e.* corresponding series are equal term by term. It is well known [16] that $s_i = \sum_{w \in A^*} f_i(w)w$, where $f_i(w)$ is the number of distinct leftmost derivations of w starting from x_i. Under the mapping that sends any symbol from A to the same symbol, say x, we obtain the generating series for a non-terminal x_i:

$$f_i = \sum_{n=0}^{\infty} \sum_{|w|=n} f_i(w)x^n. \tag{4}$$

Let $f_i(n) = \sum_{|w|=n} f_i(w)$. Then the above equation can be rewritten as:

$$f_i = \sum_{n=0}^{\infty} f_i(n)x^n. \tag{5}$$

Suppose that $x_1 = S$, where S is the starting symbol of G. Then f_1 is called the generating series of G. If G is unambiguous, then $f_1(n)$ gives the number of words of length n in G. We denote by $[x^n]f$ the n-th coefficient of f, *i.e.* $[x^n]f = f(n)$.

Let ϕ be the morphism defined by

$$\begin{aligned} \phi(\lambda) &= 1, \\ \phi(a) &= x \qquad \forall a \in T, \\ \phi(x_i) &= f_i \qquad x_i \in N. \end{aligned} \tag{6}$$

Let $x_i \to v_{i1} \mid \cdots \mid v_{ik}$ be the set of productions associated to x_i. Then f_i can be obtained as the solution of the following system of equations:

$$f_i = \sum_{j=1}^{k} \phi(v_{ij}). \tag{7}$$

For a regular grammar G the system (7) becomes linear. By considering a finite automaton $\mathcal{A} = (V, \mathbb{Q}, q_0, Q_f, \delta)$ equivalent to G we obtain that system (7) corresponds to the following system (recall that x is considered as a constant)

$$Q = xMQ + F. \tag{8}$$

where,

- $Q = [q_1 \dots q_n]^t$, $q_i \in \mathbb{Q}$, $1 \leq i \leq n$ is the vector containing all states.
- $F = [a_0 \dots a_n]^t$, is the final state characteristic vector, *i.e.* , $a_i = 1$ if q_i is a final state and 0 otherwise.
- M is the transfer matrix of the automaton $\mathcal{A}$, *i.e.* , the incidence matrix of the graph represented by $\mathcal{A}$ with negative values replaced by zero.

We remark that in the case of a regular language it is also possible to count the number of words of length n by summing the columns corresponding to the final states of the n-th power of the transfer matrix of the corresponding automaton:

$$f_i(n) = \sum_{q_j \in Q_f} (M^n)_{i,j}. \tag{9}$$

It is known that the generating series f for a regular language is rational. That implies that there exist a finite recurrence $f(n) = \sum_{j=1}^{k} a_j f(n-j)$, $k > 0$, $a_j \in \mathbb{Z}$ which holds for large n.

Example 1. Considering the regular language L_I recognized by the following automaton

$$\begin{tikzcd}
 & & q_2 \arrow[dr, "1"] & & & \\
 {} \arrow[r] & q_0 \arrow[ur, "0"] \arrow[rr, "1"] & & q_1 \arrow[r, "0"] & q_3 \arrow[l, bend left, "1"] \arrow[r, "0"] & q_4 \arrow[ll, bend right, "1"']
\end{tikzcd}$$

Then the final state characteristic vector F of this automaton is defined by $F = [0, 1, 0, 1, 0]^t$ and the transfer matrix M by

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

The corresponding system (8) of linear equations has the following solution

$$\begin{aligned}
q_0 &= \frac{x^3 + 2x^2 + x}{1 - x^2 - x^3}, \\
q_1 &= \frac{x + 1}{1 - x^2 - x^3}, \\
q_2 &= \frac{x^2 + x}{1 - x^2 - x^3}, \\
q_3 &= \frac{x^2 + x + 1}{1 - x^2 - x^3}, \\
q_4 &= \frac{x^2 + x}{1 - x^2 - x^3}.
\end{aligned} \tag{11}$$

We can expand q_0 to obtain $q_0(n)$ $(= [x^n]q_0)$,

$$q_0 = x + 2x^2 + 2x^3 + 3x^4 + 4x^5 + 5x^6 + 7x^7 + 9x^8 + \dots \qquad (12)$$

The coefficients of the above series give the number of words of the corresponding length. For example, there are 9 words of length 8 in L_I.

It is not difficult to verify that the obtained coefficients $[x^n]q_k$, $0 \leq k \leq 4$, of the corresponding power series are particular cases of the Padovan sequence $q_k(n) = q_k(n-2) + q_k(n-3)$, $n > 3$, with the following starting values:

k	$q_k(0)$	$q_k(1)$	$q_k(2)$
0	1	1	2
1	1	1	1
2	0	1	1
3	1	1	2
4	0	1	1

3 DESIGN OF THE SIMULATOR

3.1 Formal part

As it was detailed in section 1, there are several features of a static P system which define it: topology, type of objects, rules, derivation mode and how a multiset of rules is chosen from the *Appl* set. The formal framework lets us remove from this list the topology, type of objects and rules. Hence we must only worry about the application of the rules. In this section, we consider a (static) P system, Π, of any type evolving in any derivation mode, chosen a multiset of rules in a non-deterministically way. The main idea for the construction of a fast simulator is to avoid the computation of the set $Appl(\Pi, C, \delta)$ and to compute R, the multiset of rules to be applied directly. In this article we are interested in algorithms that permit to perform this computation on FPGA in *constant* time. We remark that, in a digital FPGA circuit synchronized by a global clock signal, in one cycle of FPGA it is possible to compute any function whose implementation has a delay which does not exceed the period of the global clock signal. A pipeline using arithmetical operations and, in general, any combinatorial and sequential asynchronous subsystems, are usually included in this group.

In order to simplify the problem we split it into two parts corresponding to the construction of the following *recursive* functions:

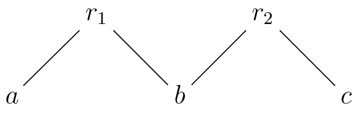
$NBVariants(\Pi, C, \delta)$**:**
gives the cardinality of the set $Appl(\Pi, C, \delta)$.

$Variant(n, \Pi, C, \delta)$**,** where $1 \leq n \leq NBVariants(\Pi, C, \delta)$:
gives the multiset of rules corresponding to the n-th element of some initially fixed enumeration of $Appl(\Pi, C, \delta)$.

It is clear that if each function is computed in constant time, then the multiset of rules to be applied can also be computed in a constant time. In what follows, we will discuss methods for the construction of these two functions for different classes of P systems.

In the following we will need the notion of the rules' *dependency graph*. This is a weighted bipartite graph where the first partition U contains a node labeled by a for each object a of Π, while the second partition V contains a node labeled by r for each rule r of Π. There is an edge between a node $r \in V$ and a node $a \in U$ labeled by a weight k if $a^k \in lhs(r)$ (and $a^{k+1} \notin lhs(r)$).

Example 2. Considering a P system Π_1 having two rules $r_1 : ab \to u$ and $r_2 : bc \to v$. These rules have the following dependency graph:



Let N_a, N_b and N_c be the number of objects a, b and c in a configuration C. We define

$$\begin{aligned} N_1 &= \min(N_a, N_b),\\ N_2 &= \min(N_b, N_c),\\ N &= \min(N_1, N_2). \end{aligned} \tag{13}$$

Suppose that Π evolves in a maximally parallel derivation mode. Then the set $Appl(\Pi, C, max)$ can be computed as follows:

$$Appl(\Pi, C, max) = \bigcup_{p+q=N} \left\{ r_1^{p+k_1} r_2^{q+k_2} \right\}, \tag{14}$$

where $k_j = N_j \ominus N$, $1 \leq j \leq 2$, where $\ominus$ is the positive subtraction operation*. The dependency of r_1 and r_2 is captured by condition $p + q = N$, considering p and q are greater than N, while k_j cover those situations in which $N_j > N$.

From this representation it is clear that $NBVariants(\Pi, C, max) = N+1$, which can be computed in constant time on an FPGA.

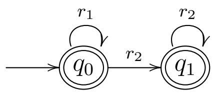
The $Variant(n, \Pi, C, max)$ function can be defined as the n-th element in the lexicographical ordering of elements of $Appl(\Pi, C, max)$ and it has the following formula

$$Variant(n, \Pi, C, max) = r_1^{N-n-1+k_1} r_2^{n-1+k_2}. \tag{15}$$

We remark that the above formula can also be computed in constant time using an FPGA.

* : $a \ominus b$ is equal to $a - b$ when $a > b$ and 0 otherwise.

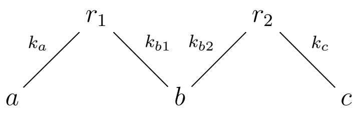
We could obtain the $NBVariants$ formula using formal power series. In order to do this we observe that the language $\cup_{N>0}L_N$, where $L_N = \{r_1^p r_2^q \mid p+q = N\}$ is regular. Moreover, it holds that $L_N = r_1^* r_2^* \cap A^N$, with A being the alphabet $\{r_1, r_2\}$. Below we give the automaton A_1 for the language $r_1^* r_2^*$.



The transfer matrix of this automaton is $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ and the final state characteristic vector is $[1, 1]^t$. Using Equation (8) this yields the generating function for L_N: $q_0 = \frac{1}{(1-x)^2}$. It is easy to verify that $[x^n]q_0 = n + 1$.

We modify the previous example by considering weighted rules.

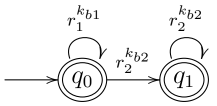
Example 3. Considering a P system Π_1 having two rules $r_1 : a^{k_a} b^{k_{b1}} \to u$ and $r_2 : b^{k_{b2}} c^{k_c} \to v$. These rules have the following dependency graph:



Let N_a, N_b and N_c be the number of objects a, b and c in a configuration C. We define

$$\begin{aligned} N_1 &= \min([N_a/k_a], [N_b/k_{b1}]), \\ N_2 &= \min([N_b/k_{b2}], [N_c/k_c]), \\ N &= \min(N_1, N_2), \\ \bar{N} &= \min(k_{b1}N_1, k_{b2}N_2). \end{aligned} \tag{16}$$

Supposing that Π evolves in a maximally parallel derivation mode. Let A_2 be the automaton recognizing the language $(r_1^{kb1})^*(r_2^{kb2})^*$.



Let $L'_N = A_2 \cap A^N$ ($A = \{r_1, r_2\}$). Then it is clear that

$$Appl(\Pi, C, max) = \bigcup_{pk_{b1}+qk_{b2}=\bar{N}} \left\{ r_1^{p+k_1} r_2^{q+k_2} \right\}, \tag{17}$$

where $k_1 = k_a(N_1 \ominus N)$, $k_2 = k_c(N_2 \ominus N)$.

The transfer matrix of A_2 (considering the weights) is $\begin{pmatrix} k_{b1} & k_{b2} \\ 0 & k_{b2} \end{pmatrix}$ and the vector $F = [1, 1]$. This gives the following generating function for A_2:

$$q_0 = \frac{1}{(1 - x^{k_{b1}})(1 - x^{k_{b2}})}. \tag{18}$$

The coefficients $[x^n]q_0$ can be obtained by the recurrence

$$a(n) = a(n - k_{b1}) + a(n - k_{b2}) - a(n - k_{b1} - k_{b2}); n \geq k_{b1} + k_{b2}. \tag{19}$$

The initial values are given by the following cases (we suppose that $k_{b1} \geq k_{b2}$):

$$\begin{cases} 1, & n = 0, \\ 0, & 1 \leq n \leq k_{b2} - 1, \\ 1, & k_{b2} \leq n \leq k_{b1} - 1 \text{ and } n = 0 \pmod{k_{b2}}, \\ 0, & k_{b2} \leq n \leq k_{b1} - 1 \text{ and } n \neq 0 \pmod{k_{b2}}, \\ 2, & k_{b1} \leq n \leq k_{b1} + k_{b2} \text{ and } n = 0 \pmod{k_{b2}} \text{ and } n = 0 \pmod{k_{b1}}, \\ 1, & k_{b1} \leq n \leq k_{b1} + k_{b2} \text{ and } n = 0 \pmod{k_{b2}} \text{ or } n = 0 \pmod{k_{b1}}, \\ 0, & k_{b1} \leq n \leq k_{b1} + k_{b2} - 1 \text{ and } n \neq 0 \pmod{k_{b2}} \text{ or } n \neq 0 \pmod{k_{b1}}. \end{cases} \tag{20}$$

Now we concentrate of the function $Variant$. If the set $Appl(\Pi, C, \delta)$ is regular, then we can use the following algorithm to compute $Variant(n, \Pi, C, \delta)$. Let $A(\Pi, C, \delta) = (Q, V, q_0, F)$ be the automaton corresponding to the language defined by rules joint applicability and let s_j, $q_j \in Q$ be the generating series for the state q_j.

Algorithm 1.

1. In the initial situation, the current state is q_0, and the other variables take the following values: $step = 0$, $nb = s_0(n)$, $out = \lambda$.
2. If $step = n$ then the system stops.
3. Otherwise, let $\{t : (q_i, a_t, q_{j_t})\}$, $1 \leq t \leq k_i$ be the set outgoing transitions from q_i, the systems computes $S(k) = \sum_{m=1}^{k} s_{j_m}(n - step)$. We put by definition $S(0) = 0$. Then, there exists k such that $S(k) \geq nb$ and there is no $k' < k$ such that $S(k') > nb$.
4. nb and out variables are updated with the following values: $nb = nb - S(k-1)$ and $out = out \cdot a_k$.
5. The system goes to step 2.

The main idea of this algorithm is to compute the n-th variant using the lexical ordering of transitions using an algorithm similar to the computation of the number written in the combinatorial number system. Being in a state q and looking for a sequence of applications of k rules we will use the transition $t : (q, r, q')$ (and add r to the multiset of rules) if the transition t is the first in the lexicographical ordering of transitions having the property that the number of words of length $k-1$, that can be obtained using all outgoing transitions from state q, that are less or equal than t is greater than n.

3.2 Hardware implementation

The goal or design key for the implemented arquitecture is to achieve a good balance between flexibility, scalability, extensibility and performance. From the point of view of a hardware designer, one of the major drawbacks about implementation and/or simulation of P systems is not their inherent parallelism, which is already an important challenge, but the great wide of types, consequence of being an oriented-machine model. As consequence, a machine must be generated for each instance of the problem, what makes unfeasible using non-reprogrammable hardware, as custom hardware or Application-Specific Integrated Circuit (ASIC). In that way, FPGAs are the only alternative in hardware to implement this computational model. FPGAs contain lots of reprogrammable logic blocks and interconnections. Users can change the hardware design a number of times limited only by the number of writing supported by the store technology of these devices. Thus, users obtain all advantages associated to reprogram, although at the cost of performance, if compared to ASIC or custom hardware. When a design is implemented using FPGAs, finding a path which communicates two logic blocks is usually the task where speed, *i.e.* performance, is compromised. As a result, modular designs which minimize long paths between logic components are the ones which best fit in this kind of technology. Our design is based on layers with interfaces clearly defined. Each layer is a block which performs a main task of the algorithm, and it only communicates with the previous layer, whose outputs are its inputs, and next layer, which receives its outputs.

In order to design the simulator, the graph of dependencies between rules has been chosen as starting point to model P systems. This approach reduces complexity, because of deleting some elements, like membranes and, in consequence, the hierarchical structure of them. Objects and rules are the only elements which have been having in mind to model the system. Moreover, the implementation is based on the mathematical foundations described in the previous section, following a division of tasks, which assures enough encapsulation to achieve a design with a right flexibility and performance. The objects are explicitly represented using registers, but it is not the case for the rules. Their logic is distributed along most of the components, thus there is no correspondence between a rule and a hardware core.

An execution of a P system consists of running iterations until it reaches a halting condition. At each iteration there is a set of operations to be carried out in order to obtain the next configuration. To implement the simulator, these tasks have been divided in the following stages:

1. *Persistence stage*: Obviously, it is necessary to save the states for which the

system goes through.

2. *Independent stage*: How *Appl* is built and how a multiset of rules is selected from it depend on the type of the P system. Whereas there are some auxiliary operations which are independent of this computation and common to most of the types: according to the rules and to the multiplicity of each type of object in the system, it is needed to know the maximum number of times that each rule can be applied without considering no others.
3. *Assignment stage*: In this stage the system chooses which rules will be applied (and how many times).
4. *Application stage*: In this level, the selected rules are applied, computing new values for multiplicity of objects.
5. *Updating stage*: In this stage, the current configuration is updated which the result of the computation of the previous stage.
6. *Halting stage*: Finally, the system has to check if the halting condition has been reached.

The simulator is divided into six blocks, two of them are dedicated to the control of it and input/output interface. So, all specific functionality of the P systems is achieved by four blocks, which follow the principle of encapsulation. All blocks take only one clock cycle to perform their tasks, except for *assignBlock*, which requires two cycles. In consequence, the simulator is able to compute and apply a transition and save the new configuration in only five clock cycles. In order to simplify the explanation, the blocks which compound the design are described following the functional division commented above (Fig. 1).

persistenceBlock :

Its goal is to persist the current configuration of the P system, update it and check adult halting condition. Hence, it implements persistence, updating and a portion of halting stages. Concerning to the two first ones, it is independent of the type of the system, being only dependent of the number of different objects and their maximum multiplicities. There is one register per type of object, that saves its current multiplicity as an unsigned integer, because of it is not possible to have a negative number of any objects. An adder performs the operation of updating the current configuration. Because of the multiplicity can decrease, this arithmetical component uses signed integers, making the conversion of the value saved in the register from unsigned to signed integer. This block receives, for each object, the differences between current and next multiplicities, and it sends to the next module the current number of objects in the system.

Although the major principle which guides the design is the modularity, some logic related to checking halting condition must be done in this block. There are several halting conditions, and which is used depends on the type of the chosen system. If this condition is reached when the state of the system does not change, it is necessary to compare the current and new multiplicities. It is

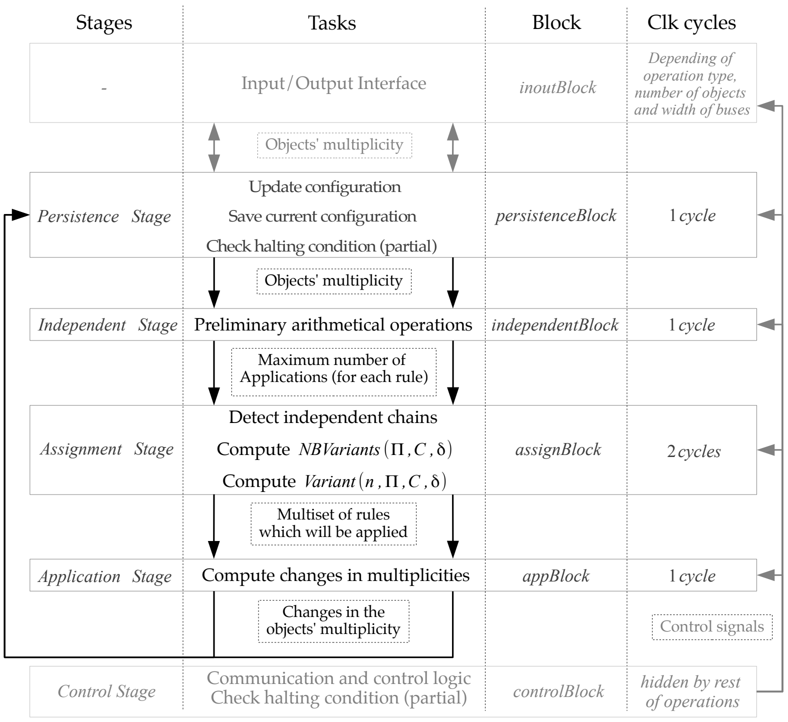

Stages
Tasks
Block
Clk cycles
-
Input/Output Interface
inoutBlock
Depending of operation type, number of objects and width of buses
Objects' multiplicity
Persistence Stage
Update configuration
Save current configuration
Check halting condition (partial)
persistenceBlock
1 cycle
Objects' multiplicity
Independent Stage
Preliminary arithmetical operations
independentBlock
1 cycle
Maximum number of Applications (for each rule)
Assignment Stage
Detect independent chains
Compute NBVariants(Π, C, δ)
Compute Variant(n, Π, C, δ)
assignBlock
2 cycles
Multiset of rules which will be applied
Application Stage
Compute changes in multiplicities
appBlock
1 cycle
Changes in the objects' multiplicity
Control signals
Control Stage
Communication and control logic
Check halting condition (partial)
controlBlock
hidden by rest of operations

Fig. 1. Overview of the architecture. This illustration shows the main blocks and the flow of information between them.

done using a comparator and a logic *and gate* which indicates to the unit control when the system fits it.

independentBlock :
It receives as input the number of objects of the current configuration from *persistenceST* and implements the independent stage functionality. It is, in consequence, an arithmetical component. In that sense, the Xilinx IP Core xilinx.com:ip:mult_gen:11.2 has been used in order to implement divisions and multiplications, being configured with a latency of 1 cycle. Although this operation is independent of the derivation mode, the fact of considering it can help to optimize the design. So, instead of giving the maximum number in absolute terms to the next block, this one usually gives this parameter in the context of the derivation mode (see section 1 for more details).

assignBlock :
This block computes the assignment stage, so it is dependent of the derivation mode and how a multiset of rules is chosen to be applied. Moreover, and as it has been previously detailed, these functionalities are performed through the implementation of the automaton which recognizes the regular language associ-

ated to dependency graph of rules. In consequence, a third dependence should be have in mind: the resource's consumption of the rules. According to hardware design, it is not possible to give a general implementation. However, we give an overview about the architecture of this block and we refer to Section 4.1 for a concrete implementation.
This unit receives the maximum number of times which a rule can be applied without considering the rest, and it gives the number of applications of each rule, *i.e.* the multiset of rules which will be applied to evolve to the next configuration, to the next one. It contains one sub-block per rule which implements the logic of the automaton. Interconnections between these components are based on design keys and propagation concepts: each sub-block is only connected to those ones associated to rules which have a direct dependence, being similar to a daisy chain network topology. Two additional components are required: the non-deterministic block (*ndBlock*) and a small unit control. The last one controls the computation of this block, because of it is usually the most complex task and it takes more than one clock cycle. It is also connected to all sub-blocks. The *ndBlock* gives to the system the non-deterministic behaviour. Its implementation and connection with the other sub-blocks depend on the necessities of the system, but the most basic form is a pseudo-random number generator. The design of this block is based on a LFSR, whose width and taps depend on the number of rules and dependencies among them.

appBlock :
It performs the application stage. It is an arithmetical component which contains adders and multipliers (same IP Core and configuration that is used in the *independentBlock*) according to the rules. It gives the number of objects which must be added to current multiplicities to evolve into the next configuration. Although other blocks (*persistenceBlock* partially) use unsigned integers in order to save hardware resources, this one operates with signed integers, because some objects can be consumed (negative number) or produced (positive number).

controlBlock :
It is required to provide communication and control logic. Control is implemented using a finite state machine, which requires five states, and it generates all control signals. This block contains the rest of the logic associated to halting conditions and its complexity depends on the type of P system. The most frequent conditions are executing a fixed number of transitions; when, given a configuration, the *Appl* set is empty; or when the configuration does not change, even when *Appl* is not empty.

inoutBlock :
Besides the previous cores, an additional block is required to provide communication with the computer. It is strictly linked to software which runs in the computer. We are developing a new software which is able to be connected to a wide range of simulators, using a serial port as a first approach. Although, it is not complete yet. In consequence, some debug cores are used to control

execution and to get results.

4 EXAMPLE OF SIMULATOR CONSTRUCTION

Once general architecture and the ideas (modelling of P Systems applying formal framework) and algorithms on which it is based have been presented in previous sections, we used the following example to illustrate the FPGA implementation for our ideas. We considered multiset rewriting rules working in set-maximal mode (*smax*). This mode corresponds to the maximally parallel execution of rules, but where the rules cannot be applied more than once. This mode can be formally defined as follows (where *asyn* is the asynchronous mode [4] and $\mathcal{R}$ is the set of all rules):

$$\begin{aligned} S_1 &= \{R \in \mathit{Appl}(\Pi, C, \mathit{asyn}) \mid |R|_{r_j} \leq 1, 1 \leq j \leq |\mathcal{R}|\}, \\ \mathit{Appl}(\Pi, C, \mathit{smax}) &= \{R \in S_1 \mid \text{there is no } R' \in S_1 \text{ such that } R' \supset R\}. \end{aligned} \tag{21}$$

We remark that *smax* mode corresponds to min_1 mode [4] with a specific partition of rules: the size of the partition is $|\mathcal{R}|$ and each partition p_j contains exactly one rule $r_j \in \mathcal{R}$.

Considering now a multiset rewriting system (corresponding to a P system with one membrane) evolving in *smax* mode. To simplify the construction we consider rules having a dependency graph in a form of chain without weights.

$$\begin{tikzcd}[column sep=small]
 & r_1 \arrow[dl, dash] \arrow[dr, dash] & & r_2 \arrow[dl, dash] \arrow[dr, dash] & & \dots \arrow[dl, dash] \arrow[dr, dash] & & r_n \arrow[dl, dash] \arrow[dr, dash] & \\
a_0 & & a_1 & & a_2 & & a_{n-1} & & a_n
\end{tikzcd}$$

Let N_{a_i} be the number of objects a_i in configuration C. The number of variants of applications of a chain of rules $r_1, \dots, r_k$ to the configuration C in *smax* mode is denoted by $NBV([r_1, \dots, r_k], C)$, $k > 0$. We remark that for a P system Π having the set of rules $\mathcal{R}$, $NBVariants(\Pi, C, smax) = NBV(\mathcal{R}, C)$.

It is possible to distinguish 3 cases with respect to the number of objects N_{a_i}, $0 \leq i \leq n$ (considering that $0 \leq s \leq i \leq e \leq n$):

$N_{a_i} = 0$. Then the two surrounding rules (r_i and r_{i+1}) are not applicable. In this case the parts of the chain at the left and right of a_i are independent, so the number of variants is a product of corresponding variants:

$$NBV(r_s, \dots, r_e, C) = NBV(r_s, \dots, r_{i-1}, C) * NBV(r_{i+2}, \dots, r_e, C) \tag{22}$$

$N_{a_i} > 1$. As in the previous case the chain can be split into two parts because both rules r_i and r_{i+1} can be applied:

$$NBV(r_s, \dots, r_e, C) = NBV(r_s, \dots, r_i, C) * NBV(r_{i+1}, \dots, r_e, C) \tag{23}$$

$N_{a_i} = 1$. In this case r_i and r_{i+1} are in conflict.

Now let us concentrate on the last case. Without loss of generality we can suppose that $N_{a_i} = 1$, $0 \le i \le n$. We remark that the language of binary strings of length n corresponding to the joint applicability vector of rules $r_1, \dots, r_n$ coincides with the language L_I from Example 1. Therefore the number of possibilities of application of such a chain of rules of length n is equal to $NBV(r_1, \dots, r_n, C) = [x^n]q_0$, *i.e.* , $q_0(0) = 1$, $q_0(1) - 1$, $q_0(2) = 2$ and $q_0(n) = q_0(n-2) + q_0(n-3)$, $n > 3$.

Hence in order to compute $NBVariants(\Pi, C, smax)$ we first split the chain into $k > 0$ parts of length n_j according to the multiplicities of objects and compute the NBV function for each part using the decomposition above.

The function $Variant$ for each part can be computed using Algorithm 1.

The next section gives more details on the implementation of the above algorithms on FPGA.

4.1 Implementation details

Starting from general architecture, there are only two blocks which can be affected by the implementation of a new type of P system: *controlBlock* and *assignBlock*. The first modification is originated by the halting conditions, and its impact is reduced to *controlBlock*. In this case, the halting configuration is reached when the configuration of the system does not change, even being the *Appl* set not empty. Attending to *assignBlock*, the derivation mode of the P system changes the way in which P system evolves, being needed to modify it. These situations are explained in Section 3.2, so we will not give more details about it.

The implementation of *assignBlock* depends on the selected evolving mode, *smax* mode in this case. According to algorithms explained in previous sections, it has to perform the following steps:

Algorithm 2.

1. The input chain is split into k parts as it is described in section 4.
2. For each part.
   (a) The system computes $NBVariants(\Pi, C, smax)$. For this purpose, algorithms detailed in 2.1 and 4 are used.
   (b) The value of n, which indicates which combination will be chosen (n-th element), is obtained. It is worth to note that his domain is from 0 to $NBVariants(\Pi, C, smax) - 1$, so, considering that the random number generator produces numbers in binary format, the following correction is sometimes needed in order to obtain it.
      i A random number, rn, is generated by the random number generator, where
      $$0 \le width(rn) \le \lceil \lg_2(NBVariants(\Pi, C, smax)) \rceil$$

ii If $rn < NBVariants(\Pi, C, smax)$ then $n = rn$.
Otherwise, $n = rn + NBVariants(\Pi, C, smax)$.

(c) Compute $Variant(n, \Pi, smax)$, according to algorithm 1.

We note that the computation of $NBVariants(\Pi, C, smax)$ uses a subset of operations needed to compute $Variant(n, \Pi, smax)$, moreover, these operations can be done in parallel with the generation of the random number n, necessary to compute $Variant(n, \Pi, smax)$. Hence, this stage can be performed in 2 clock cycles by dividing operations into two sets, called *left* and *right propagation* respectively. As it is shown by Fig. 2, left propagation is the first to be executed. In this sub-stage, steps 2.a and 2.b.i of algorithm 2 are computed from the last rule to the first one. Right propagation, which is compound by steps 2.b.ii and 2.c, is executed in opposite way in the next clock cycle. One advantage of this approach is that it is not necessary to divide, implicitly, the chain of rules in k parts, deleting a step of the algorithm which let us reduce the number of required cycles from three to two. This logic is implemented, explicitly, by signals *prevIsDep* and *chainStateSignal*. After this stage, all rules have a random multiplicity assigned.

In consequence, the architecture of this block is divided into one sub-block per rule (Fig. 2), which implements the operations required in order to obtain the number of applications of its associated rule (*left* and *right propagation*). The interconnections between these components are based on design keys and propagation concepts: a sub-block is only connected to blocks located on its right and left.

Because of *assignBlock* is the most critical component in order to achieve the maximum performance of the simulator, all operations which are required to compute $NBVariants(\Pi, C, smax)$ and $Variant(n, \Pi, smax)$ are defined recursively and are pipelined for the sake of powerful. Hence, each sub-block associated to the n-th rule computes, asynchronously, the number of times which its associated rule will be applied, based on values obtained by previous block. It permits to execute all operations in only two cycles, one for left propagation and another for right propagation, in contrast to synchronous version, which requires, at least, n cycles, due to dependencies between rules and, in consequence, between operations for computing $Variant$ and $NBVariants$.

4.2 Experimental results

Experimental results are divided into two tests. In the first one, the accuracy of the results generated by the simulator has been proved. Right after, we have obtained some results about the limitations of the physical hardware (FPGA) regarding size of the P system, and the relation performance-area required by the simulator in the device.

The P systems are the same in both tests, only modifying its size (number of rules and objects). All of them consider rules whose dependency graph forms a chain, the difference being in the right-hand side. Using these four types of dependences between rules let us check the accuracy of the results, because their design makes

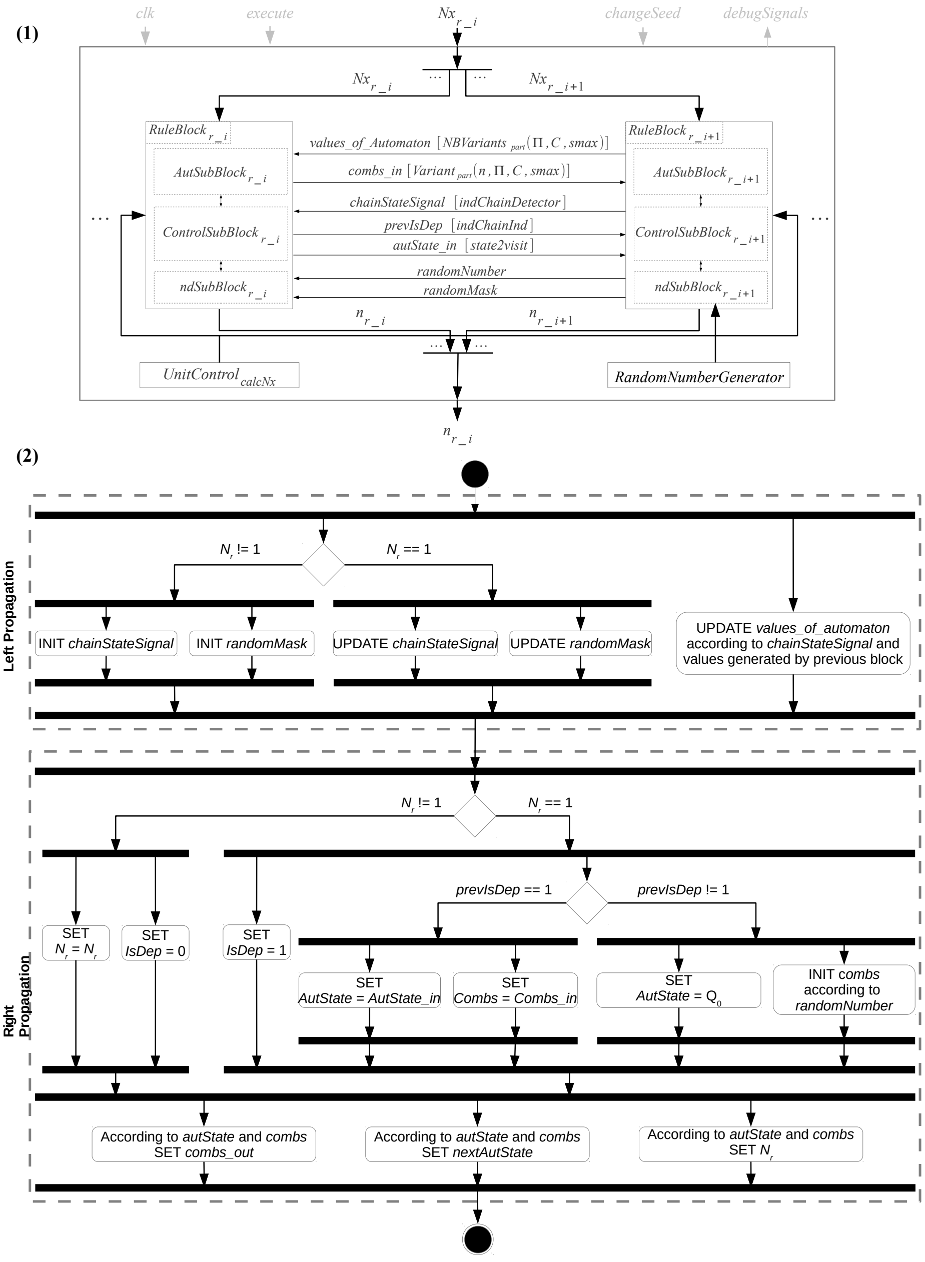


Fig. 2. Details of sub-blocks which compound the *assignRule* block. Flow of information between sub-blocks in left and right propagation is shown at the top of the figure (1). Below it, the algorithm is detailed using UML notation (2).

their behaviour predictable. We consider four P systems with the alphabet $O = \{o_0, \dots, o_N\}$, $N > 0$ and having the following rules (we consider index operations modulo $N + 1$):

- System 1 (circular)

$$r_i : \begin{cases} o_{i-1}o_i \to o_io_{i+1} & 1 \leq i < N-1, \\ o_{N-1}o_N \to o_0o_1 & i = N. \end{cases} \tag{24}$$

- System 2 (2-circular)

$$r_i : o_{i-1}o_i \to o_{i+1}o_{i+2} \qquad 1 \leq i \leq N. \tag{25}$$

- System 3 (linear)

$$r_i : \begin{cases} o_{i-1}o_i \to o_io_{i+1} & 1 \leq i < N-1, \\ o_{N-1}o_N \to o_No_N & i = N. \end{cases} \tag{26}$$

- System 4 (opposite), $1 \leq i \leq N$

$$r_i : \begin{cases} o_{i-1}o_i \to o_io_{i+1} & i \bmod 2 = 0, \\ o_io_{i+1} \to o_io_{i-1} & \text{otherwise} . \end{cases} \tag{27}$$

The target circuit for executions was the Xilinx Virtex-7 XC7VX485T, although similar results have been obtained with a Xilinx Virtex-5 FXT70 FPGA. Regarding code generation and extraction of results, different P systems were generated by a Java software and this code was synthesised, placed and routed using Xilinx tools. Since the input/output interface has not been developed yet, ChipScope, a Xilinx debug tool has been used. This tool let us, synchronously, change and capture the results, *i.e.* computation results and consumption of hardware resources, directly from the FPGA.

In order to carry out accuracy tests, an initial multiplicity of all objects equal to one and values 10, 20 and 50 were considered for N parameter for accuracy test. The P systems with these sizes are manageable enough to let us check this parameter of the simulator. Values 10, 30, 50, 70, 90, 110, 150 and 200 for N parameter were considered in case of hardware test. These values are enough in order to give a general idea about the relationship between consumption of hardware resources by the simulator and the size of the P systems. Then, for each obtained system, 1024 executions of 8192 transitions have been carried out. Each execution differs from the others by the seed required by the random number generator in the initialization stage. In consequence, different values are obtained during the assignment stage, which results in different executions. As results of experiments the following values are collected: the cardinality of objects in the last configuration, the seed of the random number generator (just for internal results) and the number of steps to reach the halting configuration, in case the system reached it.

Table 1 gives some statistics concerning the experiments. For each P system (Type, Size), the table shows if the stop condition is reached and, in this case, the minimum and maximum number of required transitions to achieved it (column *Halting*). The third column shows how many different configurations which have been reached by the system. This parameter indicates the grade in which the system evolves following an equiprobable distribution. As expected, linear and 2-circular systems reach a halting configuration, while in the other two cases it can not, due to dependencies among their rules. It can be seen that the simulation of non-determinism is done correctly – in some cases all resulting configurations are different. Figure 3 shows the maximal, the minimal and the mean value of the number of different objects. Due to size of the obtained graphs, we only show the case of 10 rules, the other cases present a similar picture. It can be seen that in the case of the linear system there is a high chance to have a big value for the last object and in the case of 2-circular systems the second and penultimate objects are never present. In the case of circular systems it is possible to see an equiprobable distribution of objects, while for the opposite systems even values have a higher multiplicity. It can be easily seen that the used rules should exhibit exactly this behavior.

Table 1. Statistics concerning the executions of example systems.

Type	N	Different final conf.	Halting		
			Y/N	min	max
Circular	10	982	No	-	-
	20	1024	No	-	-
	50	1024	No	-	-
2-circular	10	161	Yes	5	89
	20	818	Yes	11	197
	50	1024	Yes	57	609
Linear	10	204	Yes	7	17
	20	944	Yes	14	29
	50	1024	Yes	50	65
Opposite	10	4	No	-	-
	20	938	No	-	-
	50	1024	No	-	-

Concerning performance tests, they only differ in the N parameter from values of the P systems. The Xilinx tools let user control the generation of final bitstream providing some parameters. In that way, three sets of parameters, with three implementation goals, have been used: one to maximize performance, another

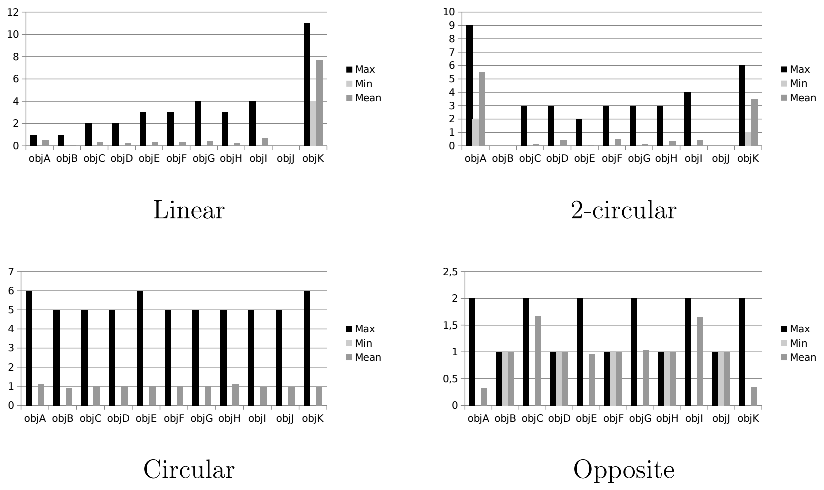
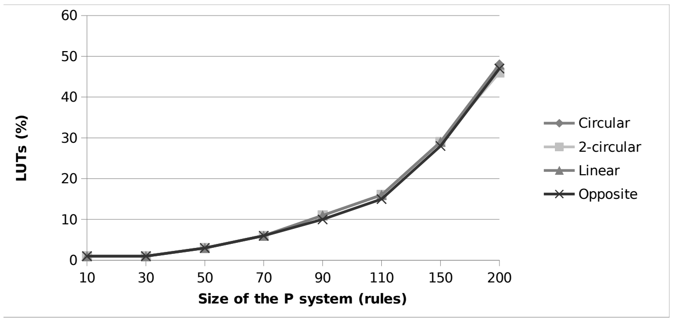


Fig. 3. Objects min/max/mean values for the experiments using 10 rules.

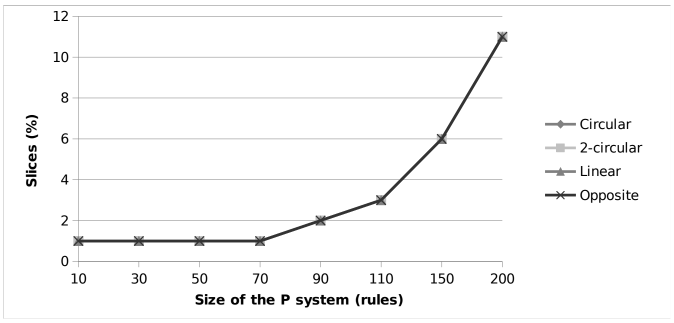
to minimize area and a balanced configuration. For each hardware design and set of parameters, values relative to performance (Table 2) and hardware resource consumption (Figure 4) were taken. In Table 2 and Figure 4 only the best results are shown: those ones achieved with performance configuration and area minimization, respectively.

Table 2 shows hardware period in ns of the system without the debug logic and Figure 4 shows resource consumption using relative values of LUTs and Slices. The implementation achieves high performance, with frequencies higher than 100 MHz, *i.e.* it allows to simulate around 2×10^7 computational steps per second. A runtime comparison between P-Lingua simulator and our hardware simulator is shown in Table 3. P-Lingua is a Java software framework developed by members of the Research Group on Natural Computing, at the University of Seville. In order to make the comparison, several opposite P systems, see Definition 27, with a number of rules between 10 and 200 have been chosen. These P systems have been converted to equivalent probabilistic P systems: they have an environment with a membrane, they follow the same dependencies between rules than opposite P Systems and they have the same number of rules and objects. The computing platform for execution was an Intel Core $i5 - 5220$ at 3 GHz, with 8 GB of RAM. The comparison shows hardware arquitecture is 5600 times faster than PLingua implementation in the worst case, when a P system with 10 rules is used, and c. 30000 times faster than Plingua in the best case, when the P system has 200 rules.

Attending to the hardware resource consumption, it only depends on the number of rules. This result is coherent with the fact that rules of all systems do not change the total number of objects and share the same dependency graph. While the period is stable, with a slightly linear increase, the resources grow exponentially, reducing the use of the simulator for really large problems, in terms of thousends of rules. Current implementation has problems for P systems with more than 1000 rules. In this case, configurations composed by several devices can be developed to move this barrier.



LUTs



Slices

Fig. 4. LUTs and Slices consumption of hardware implementation for each combination (type, sizes) in rules.

5 DISCUSSION

One of the major characteristics of the simulator is its speed, offering a constant time execution step (in terms of clock cycles). Its design is based on the method discussed in Section 3.1 where, although it is possible to design *ad-hoc* functions that describe the execution strategy of the rules, we concentrated on the cases where the multisets of rules that can be applied form a non-ambiguous context-free language. This fact allows to easily compute the generating function of the corresponding language and gives a simple algorithm for the enumeration strategy.

In this work we have focused on P systems whose rules are restricted to dependency graphs which form chains. However, the class of P systems where the set

Table 2. Period of hardware implementation for each combination (type,sizes).

Type \ Size	Circular	2-circular	Linear	Opposite
10	7.06 ns	5.63 ns	5.46 ns	7.85 ns
30	8.23 ns	8.55 ns	7.33 ns	6.99 ns
50	7.69 ns	6.71 ns	6.11 ns	6.03 ns
70	9.14 ns	7.33 ns	7.34 ns	6.87 ns
90	7.32 ns	6.38 ns	7.02 ns	7.89 ns
110	7.00 ns	7.84 ns	6.23 ns	8.30 ns
150	7.85 ns	7.42 ns	8.88 ns	7.40 ns
200	7.74 ns	8.42 ns	7.98 ns	8.35 ns

Table 3. Runtime comparison between a software simulation using PLingua, and a hardware simulation using architecture.

Size	Software	Hardware
10	0.22 s	3.927×10^{-5} s
30	0.326 s	3.495×10^{-5} s
50	0.453 s	3.017×10^{-5} s
70	0.542 s	3.435×10^{-5} s
90	0.639 s	3.945×10^{-5} s
110	0.742 s	4.151×10^{-5} s
150	0.911 s	3.7×10^{-5} s
200	1.245 s	4.174×10^{-5} s

$Appl(\Pi, C, \delta)$ corresponds to a non-ambiguous context-free language is quite big. For example, considering a set of rules forming a circular dependency graph for a system working in the *smax* mode.

$$\begin{tikzcd}
 & r_1 \arrow[dr, no head] & & r_2 \arrow[dr, no head] & & \cdots \arrow[dr, no head] & & r_n \arrow[to=2-1, no head, bend left] \\
a_0 \arrow[ur, no head] & & a_1 \arrow[ur, no head] & & a_2 \arrow[ur, no head] & & a_{n-1} \arrow[ur, no head] &
\end{tikzcd}$$

Now let C be a configuration where all these rules are applicable exactly one time (corresponding to the case 3 described in Section 4). Then the joint applicability vectors of these rules (i.e. binary strings of length n with value 1 in i-th position corresponding to the choice of rule r_i) can be described by taking the words of length

n of the following automaton

$$\begin{tikzcd}
 & & q_2 \arrow[r, "1"] & \boxed{q_1} \arrow[r, "0"] & q_3 \arrow[r, "0"] \arrow[l, bend left, "1"] & q_4 \arrow[ll, bend right, "1"'] \\
{} \arrow[r] & q_0 \arrow[ur, "0"] \arrow[rr, "1"] & & q_1 \arrow[r, "0"] & \boxed{q_3} \arrow[r, "0"] \arrow[l, bend left, "1"] & q_4 \arrow[ll, bend right, "1"']
\end{tikzcd}$$

This automaton is obtained from the automaton for the language L_I from Example 1 by adding an additional condition: if rule r_1 is chosen then r_n is not chosen and conversely.

Using similar ideas it is possible to describe with regular languages sequences of rules forming more complicated structures. For example, the following structure

$$\begin{tikzcd}
 & r_1 \arrow[dl, dash] \arrow[dr, dash] & & r_2 \arrow[dl, dash] \arrow[dr, dash] & & r_3 \arrow[dl, dash] \arrow[dr, dash] & & r_4 \arrow[dl, dash] \arrow[dr, dash] & \\
a_0 & & a_1 & & a_2 \arrow[dr, dash] & & a_3 & & a_4 \\
 & & & & & r_5 \arrow[dl, dash] & & & \\
 & & & & a_5 & & & &
\end{tikzcd}$$

can be represented as regular language over the binary alphabet if the number of symbols a_2 is known. This language can be constructed in a similar way as the language above for circular dependency. This permits to compute the function $NBVariants(\Pi, C, smax)$ by first choosing the appropriate automaton based on the value of N_{a_2} and after that computing its generating function. Clearly, this can be done in constant time on FPGA.

In a similar way it is possible to describe regular languages for the applicability of rules having the dependency graph that has no intersecting cycles.

We would like to point out another algorithm for the rule application, applicable to any type of rule dependency.

Let Π be a P system evolving in the set-maximal derivation mode. Let $\mathcal{R}$ be the set of rules of Π and $n = |\mathcal{R}|$. Let C be a configuration.

Algorithm 3.

1. Compute a permutation of rules of $\mathcal{R}$: $\sigma = (r_{i_1}, \dots, r_{i_n})$, $i_k \neq i_m$, $k \neq m$.
2. For $j = 1, 2, \dots, n$ if r_{i_j} is applicable then apply r_{i_j} to C.

The step 1 of the above algorithm can be optimized using the Fisher-Yates shuffle algorithm [17] (Algorithm P). However, the implementation of Algorithm 3 is slower than the implementation we presented in Section 3 because the computation of the rules' permutation needs a register usage, so it cannot be done in one clock cycle and it is dependent on the number of rules.

By extending Algorithm 3 it is possible to construct a similar algorithm for the maximally parallel derivation mode.

Algorithm 4.

1. Compute a permutation of rules of $\mathcal{R}$: $\sigma = (r_{i_1}, \dots, r_{i_n})$, $i_k \neq i_m$, $k \neq m$.
2. Compute the applicability vector of rules $V = (m_1, \dots, m_n)$, being m_j, $1 \leq j \leq n$ the number of times rule r_{i_j} can be applied.
3. If the vector V is null, then stop.
4. Otherwise, repeat step 5 for $j = 1, \dots, n$.
5. Compute a random number t between 0 and $V[j]$. Apply r_{i_j} t times.
6. Goto step 2.

This algorithm has similar drawbacks and the number of clock cycles it uses is at least proportional to the number of rules.

In Section 1 we simplified the model removing the forbidding and permitting conditions. Adding this feature only affects to *independentBlock*. In case of forbidding conditions, a specific comparator which checks if the objects exist should be added, this new component only differs in a *not gate* in its output, hence its logical function is the inverse of others. Then, if it returns *false*, the maximum number of applications of the rule will be set to zero. Modifications required to give support to permitting condition are similar, but checking the existence of the object, instead of inexistence.

On the other hand, P systems are a machine-oriented computational model. In consequence, each instance of a problem has a tailor-made machine which resolves it, and each machine has associated a specific hardware which simulates it. So, once we have defined an architecture which is able to simulate P systems, we need a software which generates the specific hardware design in order to program the FPGA and obtain the results. Additionally, a software which gets the results, processes it and shows it to the user will be welcome, specially for this last one.

As it is said in Section 4.2, the generation of the hardware design is done by an *ad-hoc* software written in Java. Although this solution is acceptable for testing purposes, it has several limitations, all of them derived from its poor flexibility, extensibility and modularity. In addition, there are several research lines whose straight results are simulators. Each research group deals with the same developing problems, giving different solutions and building incompatible systems, taking a high cost in effort. A P system framework can resolve, or mitigate in the worst case, all these problems. This software will resolve general problems related to end-user and developer interface, debugging and generation of code. Thus, it let developers focus on the implementation of the algorithm of its simulator. For this purpose, Model Drive Engineering (MDE) is the most suitable technology [18]. Simulators and classes of P systems will be represented as models connected by transformations: in order to generate the simulator code of a P system we need a transformation from the model of the class to which the P system belongs, to the model of the simulator. Once a instance of the simulator model is created, generating the associated code is possible using MDE.

Concerning this work, the P system framework constitutes an elegant solution to software problems. The general architecture can be seen as the model of our simulator, thus, the input is resolved by the transformation from the meta model which represents the class of P systems where the set $Appl(\Pi, C, \delta)$ corresponds to a non-ambiguous context-free language to the first one, and the hardware code generation by the transformation from the model of our simulator directly to code. In that sense, the software tool of our simulator will be highly flexible, scalable, and modular. Additionally, end-user interface will be also resolved by this framework.

Attending to performance, results show that hardware arquitecture is c. 30000 times faster than Plingua in the best case, when the P system has 200 rules. This result only refers to execution time. In that sense, if generation time is considered, *i.e.* from specification of P system to bitfile generation, first execution would take about 3 hours more*. This can be an important drawback if the P system is only executed one time. However, this kind of simulators are widely used in problems where thousands of computatios are needed. In consecuence, user should use software or hardware simulator, depending on the number of executions and size of the problem.

6 CONCLUSIONS

In this article we have introduced a fast hardware simulator for static P systems whose set $Appl(\Pi, C, \delta)$ corresponds to a non-ambiguous context-free language. Its major feature is the performance that it is possible to achieve: the hardware implementation is able to execute one transition in a constant time of 5 clock cycles, closed to the ideal value of one transition per clock cycle. In addition, the range of P systems which can be simulated is only affected by the dependences between their rules. The key point of our approach is to represent all possible applications as words of some regular or non-ambiguous context-free language. Then it is possible, using formal power series for the corresponding language, to generate the total number of possible applications and select and apply one of them. It is worth to note that the number of clock cycles is independent of any P system structure, including number of rules or types of objects. Nevertheless, a relationship between size of P systems and required hardware resources exists. According to our experimental results, the area grows exponentially, limiting really large P systems. In these situations, the hardware platform should use several devices or new architectures which combine this one with others in order to simulate the system. Besides, P systems which can be simulated only have to verify that its set $Appl(\Pi, C, \delta)$ is a non-ambiguous context-free language, accepting as input a wider range of P systems in contrast to other simulators which depend on structural elements, as type of objects.

In order to exemplify our approach, we developed several hardware simulators using a FPGA. Input P systems work in a maximal set mode with rules dependency

*: The computing platform for execution was an Intel Core $i5-5220$ at 3 GHz, with 8 GB of RAM.

graph in a form of chain. We obtained a speed greater than 2×10^7 computational steps per second. Our different tests gave right values, showing a non-deterministic behaviour in the computations and obtaining expected mean values for the outputs. Moreover, the architecture of the simulator is highly modular, enclosing all dependent logic of the type of the P system in one block, and reducing the impact of any other modification.

As a future research we plan to develop a software which generates the hardware implementation automatically from the regular language describing the rules joint applicability. This software will be integrated in a framework of P system simulators with the objective of resolving common problems in P systems development, specially those based on hardware development, and standardizing them and their related tools.

7 ACKNOWLEDGEMENTS

This work has been partially supported by the Ministry of Science and Innovation of the Spanish Government under project TEC2011-27936 (HIPERSYS), by the European Regional Development Fund (ERDF) and by the Ministry of Education of Spain (FPU grant AP2009-3625). Authors also acknowledges the support of ANR project SynBioTIC.

REFERENCES

[1] G. Păun. Computing with Membranes. *Journal of Computer and System Sciences.* **61**, 108–143 (2000).

[2] G. Păun, Y. Sakakibara, T. Yokomori. P systems on graphs of restricted forms. *Publicationes Mathematicae Debrecen*, **60**. 635–660, 2002.

[3] Gh. Păun, G. Rozenberg, A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

[4] R. Freund, S. Verlan. A formal framework for static (tissue) P systems. In *Proc. of WMC 2008* (G. Eleftherakis et al., eds.), Thessaloniki, Greece, Springer, 2007, LNCS 4860, 271–284.

[5] G. Păun. *Membrane Computing. An Introduction.* Springer-Verlag, 2002.

[6] M. Maliţa. Membrane computing in Prolog. In C.S. Calude et al., eds., *Pre-proceedings of the Workshop on Multiset Processing*, Curtea de Argeş, Romania, CDMTCS TR 140, Univ. of Auckland, 2000, 159–175.

[7] Y.Suzuki, H. Tanaka,H. On a LISP implementation of a class of P systems. *Romanian J. Information Science and Technology*, **3** (2000), 173–186.

[8] G. Ciobanu, G. Wenyuan. P systems running on a cluster of computers. In C. Martin-Vide, Gh. Paun, G. Rozenberg, A. Salomaa, eds., *Workshop on Membrane Computing 2003*, LNCS 2933, Springer, 2004. 123–139.

[9] A. Syropoulos, E.G. Mamatas, P.C. Allilomes, K.T. Sotiriades. A distributed simulation of transition P systems. In C. Martin-Vide, Gh. Paun, G. Rozenberg, A. Salomaa, eds., *Workshop on Membrane Computing 2003*, LNCS 2933, Springer, 2004, 357–368.

[10] M.A. Martinez-del-Amor, I. Perez-Hurtado, M.J. Perez-Jimenez, J.M. Cecilia, G.D. Guerrero, J.M. Garcia. Simulation of recognizer P systems by using manycore GPUs. In M.A. Martinez-del-Amor, et al. (eds.) *Seventh Brainstorming Week on Membrane Computing*, Fenix Editora, Sevilla, Spain, vol. II, pp. 45–58 (2009).

[11] B. Petreska, C. Teuscher, A Reconfigurable Hardware Membrane System. In C. Martin-Vide et al. eds., *Membrane Computing, International Workshop, WMC 2003, Tarragona, Spain, July 17-22, 2003, Revised Papers.* Lecture Notes in Computer Science, 2933, Springer, 2004, 269-285.

[12] V. Nguyen, D. Kearney, G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in Reconfig-P. *J. Log. Algebr. Program.* **79**(6), 383-396 (2010).

[13] V. Nguyen, D. Kearney, G. Gioiosa. An Implementation of Membrane Computing Using Reconfigurable Hardware. *Computing and Informatics* **27**(3), 551-569 (2008).

[14] V. Nguyen, D. Kearney, G. Gioiosa. A Region-Oriented Hardware Implementation for Membrane Computing Applications. In Gh. Paun et al. eds, *Membrane Computing, 10th International Workshop, WMC 2009, Curtea de Arges, Romania, August 24-27, 2009. Revised Selected and Invited Papers.* Lecture Notes in Computer Science, 5957, Springer, 2010, 385-409.

[15] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages.* Springer-Verlag, Berlin, 1997.

[16] N. Chomsky, M.-P. Schützenberger. The Algebraic Theory of Context-Free Languages, in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North Holland, 118–161, 1963.

[17] D. E. Knuth. The Art of Computer Programming, Volume 2 Seminumerical Algorithms. Third Edition, Addison-Wesley, 1997.

[18] J. Mukerji, J. Miller. MDA Guide V1.0.1, 2001. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`

Juan Quiros graduated of the University of Seville (Spain) in Computer Engineering in 2009. He is currently a Ph. D. candidate in Computer Engineering at the University of Seville. His research is focus on designing of hardware architectures which simulate P systems. He has expertise in MDE, hardware design, FPGA technology, design of embedded systems and development in C, java and python.

Sergey Verlan PhD in Computer Science at the University of Metz, France (2004). Habilitation in Computer Science (2010). Currently associated professor at the University of Paris Est (France). His research interests belong to the area of theoretical computer science and natural computing. He has expertise in the area of formal language theory, DNA computing, membrane computing and modeling of biological systems.

Julian Viejo received his M.S., and Ph.D. degrees in Computing Engineering from the University of Seville (Spain) in 2004, and 2011, respectively. He works as Assistant Professor at the Department of Electronics Technology of that University and has contributed several research works to international journals and conferences in the area of Digital Signal Processing and System-on-Chip design.

Alejandro Millan was born in Seville in 1975. He received his MSc in Computer Engineering in 1999 and his PhD in 2008, by the University of Seville. He works as a Professor at its Department of Electronics Technology. Since 1999, he has taught at the School of Computer Engineering and the Polytechnic School of Engineering. He is a member of its ID2 Research Group where he has participated in 23 research projects and has published more than 50 conference papers and a total of 19 journal papers.

Manuel J. Bellido received the B.Sc. degree (1987) and the Ph. D. degree (1994) in Physics from the University of Seville, Spain. He has been with the Electronics Technology Department at the same university since 1990 where he holds a post as Associate Professor. Currently he is the co-director of ID2 research group (http://www.dte.us.es/id2). Dr. Bellido has contributed tens of research works to international journals and conferences in the area of metastability, delay modelling and logic level simulation. Lately, his main interest are the embedded systems based on Open systems. he also acted as scientific consultant for a number of international journals like IEEE Transactions on Circuits and Systems, Electronics Letters, etc. He is also currently member of the Editorial Board of the Journal of Low Power Electronics.