Information Sciences 584 (2022) 360–386

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

MAREX: A general purpose hardware architecture for membrane computing

Daniel Cascado-Caballero [a,*], Fernando Diaz-del-Rio [a], Daniel Cagigas-Muñiz [a], Antonio Rios-Navarro [a], Jose-Luis Guisado-Lizar [a], Ignacio Pérez-Hurtado [b], Agustín Riscos-Núñez [b]

[a] *Department of Computer Architecture and Technology, Universidad de Sevilla, Seville, Spain*
[b] *Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Seville, Spain*

A R T I C L E   I N F O

Article history:
Received 16 October 2020
Received in revised form 13 September 2021
Accepted 24 October 2021
Available online 5 November 2021

Keywords:
Unconventional computing
Parallel architectures
Membrane computing
P systems
Hardware design

A B S T R A C T

Membrane computing is an unconventional computing paradigm that has gained much attention in recent decades because of its massively parallel character and its usefulness to build models of complex systems. However, until now, there was no generic hardware implementation of P systems. Computational frameworks to execute P systems up to this day rely on the simulation of the parallel working mechanisms of P systems by inherently sequential algorithms. Such algorithms can then be implemented as is or can be parallelized, up to a certain point, to run on parallel computers. However, this is not as efficient as a dedicated parallel hardware implementation. There have been ad hoc implementations of particular P systems for parallel hardware, but they lack to be problem-generic or they are not scalable enough to implement large P systems. In this paper, a first intrinsically parallel hardware architecture to implement generic P system models is introduced. It is designed to be straightforwardly implemented in programmable logic circuits like FPGAs. The feasibility and correct execution of our architecture has been verified by means of a simulator, and several simulation results for different P system examples have been analysed to foresee the pros and cons of this design.

1. Introduction

Membrane computing is an unconventional computing paradigm introduced by Gh. Păun [1,2] having a bio-inspired origin. It basically defines models of computation called membrane systems or P systems whose structure is composed by compartments, a.k.a. membranes or cells, containing multisets of objects. A P system can produce computations by following a set of rewriting rules that are inspired on biochemistry reactions. In other words, a P system is a machine which is able to perform calculations following the bio-inspired model of the cells. It differs from a Turing machine in that its computations are massively parallel and non-deterministic. There are many variants or types of P systems, the most common or used of which are transition [3], active membranes [4], probabilistic [5], evolutionary [6], Tissue P systems [7,8] and Spiking Neural P systems [9]. The first ones are the most basic. They apply rewriting rules allowing the creation and destruction of objects as

* Corresponding author.
E-mail addresses: danicas@us.es (D. Cascado-Caballero), fdiaz@atc.us.es (F. Diaz-del-Rio), dcagigas@atc.us.es (D. Cagigas-Muñiz), arios@us.es (A. Rios-Navarro), jlguisado@us.es (J.-L. Guisado-Lizar), perezh@us.es (I. Pérez-Hurtado), ariscosn@us.es (A. Riscos-Núñez).

https://doi.org/10.1016/j.ins.2021.10.064
0020-0255/© 2021 The Author(s). Published by Elsevier Inc.

well as destruction of membranes. The second ones include syntactic elements such as electrical charges and rules allowing the creation of membranes thus dynamically altering the structure of a P system during the computation and allowing an exponential working space on demand. Probabilistic P systems add probabilities to the execution of existing rules. The fourth ones introduce evolutionary rules to transform strings of characters. Tissue P systems define an alternative non–hierarchical structure of cells that communicate through symport/antiport rules, thus carrying out a computation. Finally, Spiking Neural P systems are inspired by networks of neurons connected by synapses. Their behaviour is very different from the rest of P systems types, and thus they will not be discussed in depth in this paper. Among the special features included in Spiking Neural P systems variants, we have e.g. regular expressions controlling application of the rules, delays, astrocytes on the synapses, etc. An interesting example is the case of Spiking Neural P Systems with Communication on Request [10,11], where no spikes are consumed or created by the rules. Instead, spikes are requested from neighboring neurons, and then moved along synapses (they only get replicated if needed, when multiple simultaneous requests are received by a neuron).

Membrane systems have been widely studied from a theoretical perspective [12–14], but also employed to build models of complex systems in a wide range of fields, like Bioinformatics [15,16], Biology and Ecology [5], Economy [17], Robotics [18–20], machine learning [21], knowledge-based problem-solving [22,23], and so on. A summary of its applications can be viewed in [24].

Up to now, several software simulators have been created, to simulate the behavior of a P system [25,26]. A programming language to define P systems called P-Lingua [27,28] has also been defined. However, given the nature of a P system, its execution in a computer that follows the Von Neumann architecture model is not efficient enough. Therefore, attempts have been made to achieve parallel emulation using GPUs [29–31]. However, these simulations try to reproduce similar results but not the real internal behavior of a P system.

With respect to P systems hardware implementations, there have also been several attempts from the very beginning of this computing paradigm. In [32,33] the most relevant are classified, compared and analyzed. However, these prototypes are usually ad hoc designs for specific types of P systems with many limitations. In addition, the input and output data specifications are also proprietary or particular.

In this paper MAREX (Membrane Architecture Rule EXecution) is defined as an architecture model of a processor capable of reproducing the operation of a basic transition P system. To validate the operation of this processor a software that fully emulates this architecture has been developed.

The paper is structured as follows. First, Section 2 provides a quick layman-oriented introduction to P systems, and Section 3 summarizes the main related works, explaining the differences with respect to MAREX. Then, Section 4 is devoted to introduce the machinery, showing the main components and algorithms. Next, a detailed view of each architecture's components is shown in Section 5. The description of the architecture is completed with Section 6, in which a simulator is presented, together with some simulation results for checking the correctness of the architecture's operation principle.

A discussion and some ideas for future work can be found in Section 7.

Finally, conclusions are presented in Section 8.

2. P systems

In order to facilitate a full understanding of the paper by a broader audience, the definition of the seminal version of P systems that has been chosen as standard in this paper, *basic transition P system*, is recalled here. Some references for interested readers are also provided, together with some comments about other P systems models which are currently being used for applications.

As it was already mentioned in the Introduction, membrane systems (a.k.a. *P systems*) are automata-like mathematical entities inspired by the structure and functioning of living cells. More precisely, in the basic definition one considers a hierarchical structure of membranes, and a multiset of symbol-objects located in the regions, which can be seen as a metaphor of biochemical molecules floating in the compartments of an eukaryotic cell. Note that a multiset is nothing else than a set where the elements can appear repeated, and they are usually represented as strings, although the order is not important (for example, a multiset with two copies of a and three copies of b can be represented as $babab$, $aabbb$, etc, but it will typically be denoted as a^2b^3). The set of symbols that can be used in a system is called the *working alphabet*, and is typically given explicitly when defining the system.

Each of the regions of the system has a set of rewriting rules associated to it, which can be seen as a metaphor of chemical reactions. In the general form, a rewriting rule is written as $u \to v$, where u stands for a multiset of objects and v is a multiset of objects with target indicators. That is, a rule associated with a region i can be triggered (the reaction can take place) if at some instant the contents of that region provide enough copies of all objects in u, and the effect of the application of the rule is that all objects in u are deleted, and all objects in v are created, and they will either remain in region i or be transferred to adjacent regions, depending on their associated target indicator. Rules can be applied several times if there are enough objects. The model also includes a special symbol acting as dissolution indicator, in such a way that if a rule having this symbol is applied, then the membrane where it was executed is dissolved: the associated set of rules is lost, and the contents (objects and inner membranes, if any) are transferred to the parent membrane. Note that the outmost membrane (known as *skin*) cannot be dissolved.

The semantics of the system dictates that rules should be applied in a *maximally parallel* way. This means that it is forbidden to skip the application of a rule provided that there are enough "available" objects (in the sense that they are not involved in another rule). In addition, the original definition includes the possibility to declare a set of *priorities* among rules, indicating that some rules cannot be executed as long as there are other rules with higher priority which are also applicable on the available objects. There is an implicit "global clock", in such a way that the computation executed by the system follows a sequence of transition steps, and on each step rules are applied featuring a two-level parallelism: all regions evolve in parallel, and on each region the selected rules to be executed (according to the maximally parallel principle) are applied simultaneously. The systems are non-deterministic, so it could happen that at a given moment there may exist several choices of multisets of rules to be applied, all of them compliant with the priority restrictions and the maximally parallel requirement. The output generated by the system is defined then as the collection of all possible outputs obtained by the computations emerging from a given initial configuration. More precisely, the output of a computation is typically defined as the multiplicity of some particular objects in the output membrane in the halting configuration, that is, when no rule is applicable in the system.

In the framework of basic transition P systems, the role of the compartments was soon proved to be irrelevant, as far as the computational power of the system is concerned. Indeed, in [2] it was proved that the class of families of sets of natural numbers generated by basic transition P systems does not change even if an arbitrary number of membranes is allowed. The intuitive idea is that one can use a renaming over the whole alphabet, adding to each symbol an index indicating the region where the object currently belongs.

On the other hand, P systems can also be interpreted as devices solving problems, slightly modifying the protocol for handling the output. First, a given instance of the problem is encoded into the initial configuration (e.g. by means of a multiset over an input alphabet), and then the solution associated to this instance can be retrieved from the halting configuration. Obviously, the non-deterministic behavior should be controlled in order to avoid undesired outputs. It is not necessary to impose deterministic designs, it suffices to guarantee a *confluent* behavior, in the sense that all computations yield the same answer, even if some intermediate steps might differ. This approach has been successfully applied to design solutions to hard problems using *P systems with active membranes*, which include division rules, inspired by the mitosis process. This ingredient makes it possible to generate an exponential number of membranes in a linear number of steps, and such space–time trade-off can be exploited to obtain "efficient" solutions (in terms of the number of computation steps of the system). However, such membrane division rules fall out of the scope of this paper, since it is unrealistic to assume that an exponential amount of hardware resources is available *a priori*.

The design of the architecture presented in this paper is not restricted to a particular variant of P systems. We follow a general-purpose approach instead, considering generic patterns for rules, and focusing on the process of detecting which rules are applicable to a given configuration (that is, the question: "are objects in the left-hand side available?") and how many times each applicable rule will be executed (recall that each copy of an object can be captured by at most one rule). In this way, our implementation could be used for any type of evolution-communication rule (not only for basic transition P systems, but also for P systems with active membranes, tissue P systems, etc).

Recently a third approach is being explored: the result of the system is not defined based on the halting configuration, but on the computation itself. This is suitable for modelling purposes, so that the evolution of the P system is interpreted as the evolution of the phenomenon that is being modelled. In this context, it is often the case that the rules have probabilities associated to them, and consequently the P system needs to be simulated multiple times so that the results collected from each run are processed together and the combined information can be statistically significant (see e.g. [5]). Further examples of real-life applications engineered within the membrane computing paradigm can be found in [24].

3. Related works

This section overviews previous works in the membrane computing literature addressing the challenge to build P systems simulators and focusing on the hardware level.

The first hardware implementation of a membrane-based system is found in [34] and was made shortly after the creation of the membrane computing paradigm. The aim of that preliminary work was the definition of a minimal hardware architecture and not a faithful classical P systems implementation. Thus, authors had to introduce some simplifications for their hardware implementation, namely: 1) rules were not applied in a maximally parallel manner but following a predefined order; 2) The P system was to be deterministic, i.e., the computational tree contains only one branch; 3) Rules inside a membrane (called micro-steps in this work) were applied sequentially. The main reason for this was that, using those authors' architecture model, a fully parallel rule execution would have been too expensive to implement as it would require a search algorithm. In this work, each membrane had registers for each type of object and each rule had its own circuit, and the membranes communicated with their parent or children membranes by means of explicit buses. This work presented also some initial implementation results of small P systems in VHDL for Field Programmable Gate Arrays (FPGAs). As far as we know, this opening work was not followed by more exhaustive hardware implementations. Merely, in [35] the author addressed and discussed some general issues related to genuine hardware realizations at the microscopic level.

In [36,37] the design of a circuit to execute parallel rules in a P system was proposed. A later paper by these authors [38] proposed an implementation using PIC microcontrollers and EEPROM flash memories interconnected by an I^2C bus. Here, each microcontroller simulated the operation of a membrane. Memories stored information about the type and number of objects existing for each membrane, and a set of buses managed the communication and synchronization between membranes. In [39], a software architecture for the control of this microcontroller oriented system was also described as a complement to the former.

To end with, in [40] some of these authors also implemented the circuitry for selecting the active rules for a specific transition P system in an FPGA. In this sense, this implementation was not focused on a parallel and flexible approach, as it read sequentially the rules previously defined in ROM memory in order to decide which ones were applicable.

Another paper (see [41]) from the same period of time proposed an implementation of a P system written in Handel-C also for FPGAs. In this model each rule of a membrane had its own implementation and all of them were managed by a central coordinator. Later in [42] this model was extended to introduce a non-deterministic algorithm for assigning objects between rules. In [43,44] these authors extended their model with messages (objects) between membranes and a better synchronization of hardware components to make it more similar to a P system.

The previous set of works resides in the following implementation paradigm. The order in which rules can be applied and objects are produced by them is divided in an artificial exogenous manner in two phases: preparation phase, where objects are assigned to rules that require them, and, updating phase, where each applicable rule updates one or more multisets of objects according to its definition. Therefore, implementing the preparation phase involves calculating for each rule the maximum number of instances that can be applied in the current computation step of the P system given (a) the current state of the multiset of objects in its region and (b) the relative priorities and requirements of the other reaction rules in its region. This implies the use of algebraic and algorithmic operations (integer division, binary tree processing, and so on), and some additional algorithmic complications like time-oriented and space-oriented conflict resolution strategies. All of these issues separate this implementation to the inherent massively parallel nature of P systems. On the contrary, the present work defines a suitable architecture for solving these issues in a natural manner.

In [45] a specific application about image segmentation in hardware is presented by means of FPGAs through the combination of P systems capable of processing images of 4x4 pixels. [46] shows an automated system that converts Python code for simulating P-Metabolic systems (a type of P systems) into VHDL code. This was one of the first works that automated the process of translating P system high level code to a hardware description code such as VHDL.

Finally, in [47] a simulator of non-deterministic static P systems was defined using formal power series, so that this simulation was able to be implemented in Field Programmable Gate Arrays (FPGA). That solution was specific for static P systems (i.e. transition P systems, not using dissolution neither division rules). Although this simulator was efficient in terms of hardware performance, it consumed resources exponentially as the P system grows in size.

In all the previous approaches for P system hardware implementations, two types of general designs are observed: rule-oriented and region-oriented. In addition, they generally have to be programmed/configured/adapted in a non-automatic way at a low level, and all of these cited implementations have scalability issues at the hardware level. Another common observed characteristic is that they mostly take into account only specific P systems (generally basic transition P systems). This greatly limits their use for other types of P systems.

Finally, most of them try to somewhat emulate the intrinsic computing behavior of P systems (object competitiveness, cooperative rules, rule priorities, etc.) in sophisticated hardware-coded algorithms or algebraic operations, instead of defining an architectural model being inherently capable of mimicking them.

Therefore, it is demonstrated that there is a need to look for an alternative hardware implementation of P system models with a more natural architecture model to the membrane computing paradigm, being as well more modular, scalable, usable, easily configurable/programmable, run-time efficient, and capable of simulating more types of P systems. This process is neither trivial nor immediate. That is why a first approach to achieve this goal consists in modeling basic elements in hardware, which are capable of accomplishing, at least, an individual membrane computation according to the above mentioned criteria.

4. Operation principle

The architecture model presented here (MAREX) is oriented towards the parallel processing of rules within a membrane. The main idea of this model is to have a set of rule units that absorb objects as they pass along them in a similar fashion as the oxygen is absorbed by the different cells of an organ when blood passes nearby. Thus, objects are pumped out from a central store and circulated to the rule units, which absorb those objects that match the left-hand side of the represented rule. From now on, in our architecture model a membrane object is to be represented by an entity called OCU (Object Container Unit) containing the object tag plus additional fields that manage its working along the membrane processor.

Basically, an OCU contains the object type and its multiplicity, which can be divided by two sub-attributes: active and queued. Active multiplicity expresses the quantity of objects of a specific type that can be processed by rules during a computation step. Queued multiplicity expresses the quantity of new objects of a specific type produced by the rules in the current computation step.

At least, four components (Fig. 1) are required in this model:

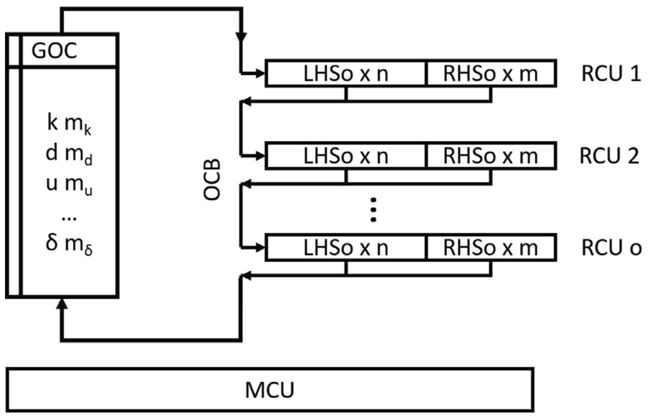

Fig. 1. The MAREX architecture model.

- MCU (Membrane Control Unit): it manages all the control logic of the membrane in a similar way that the control unit does in a CPU.
- GOC (Global Object Container): Object store that pumps out the membrane objects (or, more exactly, the OCUs) to the bus.
- OCB (Object Circulating Bus): this bus connects the GOC output to all the rule units (in a chain) and then returns to the GOC input. It continuously extracts OCUs from the GOC and sends them to the rule units. At its opposite end, the OCB collects objects from the rule units and stores them back in the GOC. In the meantime, the objects may have been processed by the rule units. In our current architecture, the OCB can be seen as a circular queue connecting different rules and the GOC.
- RCU (Rule Container Unit): Each RCU represents a membrane rule to be executed. Thus, RCUs catch OCUs from the OCB and return new elements once the rule is executed. In this architecture model, RCUs are connected to the OCB in a daisy-chain form, so that the order in which rules are attached to the OCB may have an influence on the rule priorities. This issue is discussed in more detail below.

Fig. 1 depicts the overall components of our architecture model and their main relations. GOC contains objects with their respective attributes (like multiplicities expressed as m_k for object k, m_d for object d and so on). There must also be a special object δ that will initiate the membrane dissolution. Note that this object is not needed in a P system with only one membrane, because the skin membrane cannot be dissolved. It has been included thinking about future versions of this architecture, which allow the implementation of more complex P systems where some membranes can be dissolved by connecting several MAREX processors. Further explanations about how to implement membrane dissolution in this architecture can be found at Section 7.4.

Algorithms 1 and 2 express the functioning of our model, which is summarized below.

GOC objects are launched to the OCB in a determined sequence (named *burst* from now on), starting with a special object called α and ending with another special one called Ω, as indicated in Eq. 1.

$$\alpha \rightarrow Element_1 \rightarrow Element_2 \rightarrow \ldots \rightarrow Element_m \rightarrow \Omega \tag{1}$$

OCUs between α and Ω represent current membrane objects to be processed by the P system. When there are no more OCUs to be launched by the GOC, a special object (called *NULL*) is pushed by the GOC that denotes that there are no elements left in the GOC. NULL is an object equivalent to the NOP (no operation) instruction of a traditional processor. A *NULL* object can not be captured by any rule because it is not thought to interact with them.

More precisely, after Ω is pushed into the bus, *NULL* objects will be pushed repeatedly into the bus, until Ω enters again in the GOC. This behavior guarantees that no new burst will be pushed into the bus until the object Ω of the previous burst arrives to the GOC. Thus, Ω is a terminating object that implies the end of an object burst of a specific membrane. The object Ω has been introduced with the aim of future multi-membrane implementations.

As it is shown below, these beginning and ending marks are intended for the correct rule execution and the whole operation of the P system.

RCUs contain the specification of rules, namely, those left hand side (LHS) objects to be consumed, and right-hand side (RHS) objects to be produced, by each rule.

The object gathering policy for the RCUs is a complex issue and can be implemented accordingly to the P system definition. The variety of P systems is enormous as described in the previous section, but two modes fit appropriately to the proposed architecture: *egoistic mode* and *altruistic mode*. In the first mode, the rule catches all the active multiplicity from its OCB OCU. In altruistic mode, the rule catches only the active multiplicity that it needs to perform an execution. At a first glance, egoistic mode fits well for confluent systems where a fast computation is desirable, because each rule execution is done with all the possible objects. However, if a high degree of competitiveness for several objects were present in rule definition, this mode would probably produce many computation steps where no reaction would be applied. This effect can be understood if we think about two rules with a high number of identical LHS objects. In this case, a rule reaction would be produced only if one of two RCUs caught all its LHS objects. Then, an unbiased (e.g. random) distribution of objects among the two rules implies that a reaction happens with a probability that is dramatically decreasing with the number of LHS objects. This kind of situations can be avoided by the altruistic mode, because this mode favors the parallel execution of several rules (each using only the minimal necessary objects to generate some RHS objects). In Section 5 this discussion is extended with the execution times associated to these modes.

When a rule is executed and produces new objects in its right term, the recently produced RHS objects are kept in the RCU as "queued" until an OCU passing through the circular bus has the same type of these RHS objects. Then, these produced objects are sent by the rule to the OCB and then stored in the GOC as queued objects. They are not considered until the next computation step.

When the element Ω arrives at one RCU (ensuring that all the elements of the GOC have passed in front of it) the rule would set its PURGE state if it had not captured all the LHS objects necessary to perform an execution. On this state, rule execution is not allowed. From now up to the end of the computation step, the rule flushes the objects gathered before to the OCB (when an object of the same type passes next to the rule). Flushing operation consists of adding the active multiplicity of the LHS OCU implied, to the active multiplicity of the OCB OCU, and setting to zero the active multiplicity of the LHS OCU implied.

With respect to the RHS OCUs, the flushing of the queued objects is always allowed regardless the condition of the PURGE flag.

Finally, if the element Ω arrives at one RCU that has (RHS or LHS) elements to purge, the active multiplicity of Ω is increased by one. When Ω arrives to the GOC, the computation step is prevented if the active multiplicity of Ω is greater than zero. By this mechanism, the computation step is prevented until all the rules are completely purged. Of course, each time Ω goes out to the OCB, its active multiplicity is cleared.

Rule purging allows to pass elements between rules and prevents element accumulation in the high-priority rules. This way, elements can be freed from the first rules in the OCB (nearer to the output of the GOC) to the last ones, if the first ones cannot process them.

With respect to the GOC, it continuously launches the sequence of elements until it detects that the active and queued multiplicities of all the elements do not vary between two sequences. When this occurs, a computation step has taken place. Then, GOC updates its OCU multiplicities, by adding the queued multiplicities to the active ones. This will allow the previously queued objects (which were waiting for the end of the current computation step) to be processed in the next computation step.

GOC is uninterruptedly launching sequences of objects to the bus until it detects that no new element is produced by the rules between two computation steps. This fact supposes the end of the membrane processing.

Algorithm 1:Overall architecture working

```
{Each process is preceded by the component that executes it. Those actions without a component are processed in the
  MCU.}
{$\alpha$ denotes the beginning of the Computation Step}
GOC_Launch $\alpha$
GOC_Receive_OCU
repeat
  end ← true
  something_changed ← false
  new_objects ← false
  for i = 1, ..., number_of_OCUs_in_GOC do
    GOC_Launch_OCU $O_i$
    GOC_Receive_OCU
  end for
  GOC_Launch Ω
  GOC_Receive_OCU
  repeat
    GOC_Launch NULL
    GOC_Receive_OCU {it updates variables end, something_changed}
  until GOC receives Ω
until end == true
```

Algorithm 2:Receive_OCU

```
{All of these actions are processed in the GOC.}
Get a new OCU $O$ from the bus
**if** $O.type <> \alpha$ AND $O.type <> \Omega$ **then**
  Find $i$ such as $O_i.type = O.type$
  **if** $O_i.queued <> O.queued$ OR $O_i.active <> O.active$ **then**
    $something\_changed \leftarrow$ true
    $O_i.active \leftarrow O.active$
    $O_i.queued \leftarrow O.queued$
  **end if**
  **if** $O_i.queued > 0$ **then**
    $new\_objects \leftarrow$ true
  **end if**
**else**
  **if** $O.type == \Omega$AND $O.actual == 0$ **then**
    **if**$something\_changed$ = false **then**
      {A Computation Step has occurred}
      **for**$i = 1 \ldots, number\_of\_OCUs\_in\_GOC$
        **if** $O_i.queued > 0$ **then**
    **end** $\leftarrow$ false
      **end if**
      $O_i.active \leftarrow O_i.active + O_i.queued$
      $O_i.queued \leftarrow 0$
    **end for**
    {Denotes the beginning of a new computation step}
    $GOC\_Launch\ \alpha$
    Get a new OCU $O$ from the bus (it is just a NULL object)
    **end if**
  **end if**
**end if**
```

5. Building blocks

This section presents a detailed definition of the building blocks of the proposed membrane architecture model and their operation, showing in detail the involved signals and components.

5.1. Object Container Unit

Two object attributes are considered in our P system architecture: type and multiplicity. In this way, given an object a with multiplicity 3, it is expressed with both attributes like $(a, 3)$.

As stated before, new generated objects cannot be used immediately, but after a computation step. This implies that rule execution at the current step can only operate with the existing objects at the beginning of the step. It cannot operate with the new ones produced during the actual computation step (which must wait for the next computation step). Therefore, not only must the actual multiplicity be saved for each object, but also its queued multiplicity. Queued multiplicities will be taken into account for the next computation step, by adding them to the remaining actual ones when the step is “exhausted”, that is, no more rules can be executed in the current step.

In our implementation, an Object Container Unit (OCU), which is the instance of a mathematical object, is basically a register with the following fields:

- Type: it contains the object name, that is, a unique tag in the P system.
- Active: it stores the active multiplicity.
- Queued: it stores the queued multiplicity.
- Index: it points to the position of the object within the GOC. Index is used when an OCU returns to the GOC after completing a turn along the OCB. This simplifies the implementation, since this field indicates the GOC which position this OCU must be stored in.

Finally, the operations associated to an OCU must be, at least:

- Update field values. In particular, during rule execution, those fields that indicate quantities like Active and Queued must be updated.
- Read field values.
- Reset. this operation sets field Type to *NULL* and the rest of fields to zero.

5.2. Rule Container Unit

A rule produces new objects for the membrane where it resides in and returns new produced objects to the bus. In order to accomplish such purposes, an RCU must contain the following components, which can be seen in Fig. 2:

- Rule Control Unit: it controls the working flow of the RCU and all the below elements. In addition, it must support a command-oriented interface to reprogram externally the RCU.
- LHS rule OCUs: these are OCUs specialized to the RCUs that contains two sets of fields: a) those representing the type and number of objects that the RCU needs to capture. b) those needed to specify what kind and multiplicity of objects the rule needs for the execution. These additional fields are described below. With respect to the ordinary OCU fields, Type has the same meaning, but the field *Active* saves the multiplicity of this object gathered by the rule from the current OCB OCU. When the number of objects is greater than or equal to the number needed, a flag is set to indicate this. According to the below additional fields, this can be expressed as: *if* Active $\geqslant$ Needed, *then* OK = 1.
- RHS rule OCUs: these specialized OCUs save information about the objects that the rule produces after its execution. The field *Produces* denotes the quantity of new objects that the rule produces when it is executed. With respect to the ordinary OCU fields, field Type maintains their regular meaning, but the field *Queued* saves the multiplicity of new produced objects, which are to be sent to the OCB.
- Purge: this flag indicates when the rule is forced to send its LHS elements to the OCB. It is controlled by the Rule Control Unit.
- Enable: A flag indicating that the rule can be executed.
- ExeC: This is a descending counter that limits the number of times that one rule can be executed within a computation step. It serves to control whether a rule is maximal (there is no limit for this number of times so the value of this counter is -1), minimal (the rule can be executed only once and the value of this counter is 1) or something between (for example if one rule must be executed at most two times, the value is 2). This counter is decremented each time the rule is executed.

In addition, for the specification of rules inside the RCUs, the following fields are to be provided in their LHS OCUs:

- Prob: it serves to determine the probability that an object is caught by a rule. This field contains two numbers, *min* and *max*. It performs rule competitiveness, i.e., when an object is present in two or more rules in their left-hand sides, using the following mechanism. A random number is stored in the OCB OCU when it was launched by the GOC; only if this number is between *min* and *max*, this OCB OCU can be captured by the RCU. Besides, the designer can easily arrange which is the probability that an object is captured by any of the RCUs. This is further discussed in sub-Section 5.2.3.
- Random: it contains a random number [0–1] used only in the RCUs. Its aim is the emulation of rule competitiveness for the objects circulating by the OCB. This is discussed in sub-Section 5.2.3.

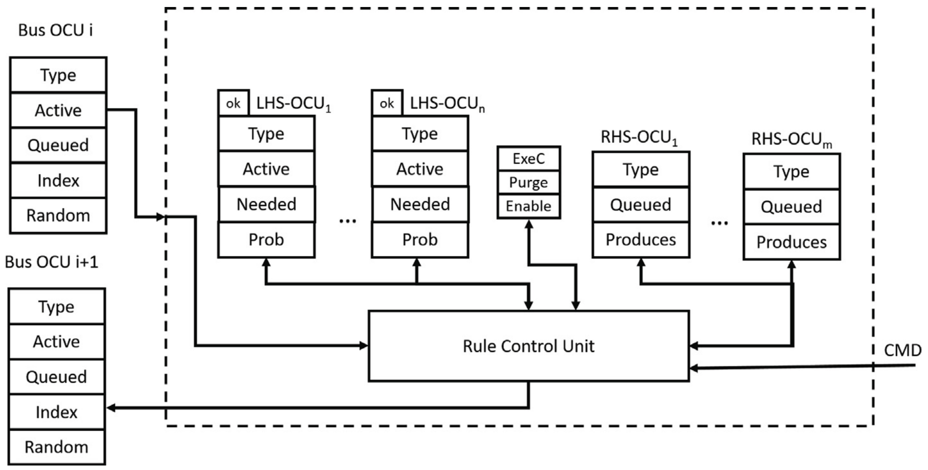

Fig. 2. RCU connected to the bus OCB. RCU is composed by all the elements within the dashed square. The RCU is connected to OCU_i and OCU_{i+1}. They are part of the OCB. This RCU is capable of implementing rules with n elements in its LHS and m elements in its RHS.

- Needed: it defines the required multiplicity of the object for each rule LHS object.
- OK: it indicates that there are enough of these LHS objects to execute this rule.

Moreover, the following fields are needed in the RHS OCUs to specify the production of objects of each RCU:

- Produces: it denotes the RHS object multiplicities for the objects produced by a rule.

5.2.1. Execution phases

The Rule Control Unit is a state machine with the following operation phases:

- Left purge: In this phase, the field *Type* of the incoming OCB OCU is compared with all the types of the LHS OCUs. That LHS OCU with the same type is selected, if any. If the flag Purge is set, the active multiplicity of such LHS OCU and the active multiplicity of the incoming bus OCU are added and written to the active multiplicity of the outgoing bus OCU. Consequently, the active multiplicity of this LHS OCU is set to zero.
- Object gathering: It compares the type field of the incoming bus OCU and compares with all the types of the LHS OCUs. The LHS OCU with the same type is selected. If *Purge* is not enabled, and field *Random* of the incoming Bus OCU is inside the interval defined by the field *Prob* of the L-OCU selected, the active multiplicity of such L-OCU is added to the active multiplicity of the incoming Bus OCU and written in the active multiplicity of the entering Bus OCU, and the active field of the outgoing Bus OCU is zeroed. This implies that the rule gathers all the objects that pass through it. If the type of element is α, ExeC is reset to its original value and the flag Purge is disabled.
- Purge evaluation: if the type of the incoming OCU equals to Ω, Purge is evaluated. In addition, this entering Ω is copied to the outgoing bus OCU to enable the purge in the rest of RCUs. Purge is committed inside a RCU if the rule has gathered some LHS objects but not all the necessary, that is, it could not be executed. Then, RCU must purge all the previously gathered objects to the OCB in order to allow the other RCUs below it to use these objects. If an RCU received the element Ω and it was already in purge, it would indicate that there would still be elements to be purged by increasing the "active" multiplicity of the outgoing Ω by 1. This informs the system that there are still objects to be purged and that the Computation Step can not be triggered. Finally, when the incoming OCU Type equals to α, Purge is disabled.
- Execution: if all the OKs on the LHS OCUs are set and ExeC is greater than zero or equal to -1, the rule produces new elements in the RHS OCUs, adding to the field Queued (of all the RHS-OCUs) as many objects as the field Produces indicates. ExeC is decremented by one.
- Right purge: in this phase, the field *Type* of the incoming GOC OCU is compared with all the types of the RHS OCUs. If it is an RHS OCU with the same type, it is selected. The queued multiplicity of such RHS OCU is incremented and then written in the queued multiplicity of the outgoing bus OCU. Then, the queued multiplicity of the above mentioned RHS OCU is set to zero. Finally, the type and the random field of the incoming bus OCU are copied to the outgoing bus OCU.

Algorithm 3:RCU Phases.

```
if Purge then
  Left Purge
else
  Object gathering
end if
Purge evaluation
Execution
Right purge
```

5.2.2. Number of rule executions per computation step and object gathering policy

ExeC is an execution counter for limiting the number of times that a rule can be executed within a computation step. *ExeC* values are set at the definition of the P system and they characterize the type of rule depending on their value:

- Minimal: the rule can be executed only once in a computation step. The value of *ExeC* is 1.
- Intermediate: the rule can be executed a finite number of times *t*; thus *ExeC* is setup to *t*.
- Maximal: the rule can be run as many times as possible. The value -1 indicates such circumstance.

A rule can be executed if *ExeC* is different from zero. If *ExeC* is different from -1, it is decreased by 1. This counter is restored to its original value at the start of the next computation step, which is indicated when the OCU α is passed by the RCU.

Another important issue is the object gathering policy followed by the RCUs. Two cases are common in P system definitions, splitting rules into two sub-types, which are implemented in our architecture model straightforwardly. These are:

- A rule gathers all the active multiplicity of the incoming OCU. The rule can be called “egoistic”.
- The rule just gathers the minimum active multiplicity of the incoming OCU that it needs to be executed only once. The rule can be called “altruistic”.

Having all the rules the same object gathering policy, in the case the P system is confluent both altruistic and egoistic rules may steer it to the same solution, but in most of the cases altruistic rules lead to longer computation steps. Here, the length of a computation step can be defined as the number of times the MCU pumps out an OCU to the OCB until the GOC receives the element Ω, which ascertains the computation step ending. On the other hand, an altruistic rule gives more opportunities to the rest of rules to be executed, thus promoting the exploration of diverse paths in the P system evolution. In conclusion, different forms of the rule’s object gathering may lead to different solutions in case of non-confluent P systems. See Section 2 for more details about this.

Given that all the rules within a membrane processor have the same type of object gathering policy, we could have altruistic or egoistic membrane processors, and even, P systems if all the membrane processors have the same type of gathering policy.

5.2.3. Rule competitiveness and rule execution conditions

As shown in Fig. 1, rule location in the OCB determines the order on which rules gather objects from this bus: these rules nearer to the GOC output would have higher probability of gathering objects than those rules nearer to the GOC input. Yet this hardware bias must be prevented by managing conveniently the RCU object gathering by means of the fields *Prob* and *Random* located in the OCB and LHS RCU OCUs. Let us recall that an RCU gathers an object from the OCB, only if $Prob.min \leqslant Random < Prob.max$, being *Random* the random number contained in the corresponding OCB OCU, and *Prob* the field containing the interval [min,max) in the LHS OCU that wants to collect an object from the OCB.

Let us suppose here that both fields are real numbers in the interval [0, 1] (notice that they would be implemented as integers in a simple hardware architecture), the former expressing the probability that an RCU gathers an object.

If a set of rules had the same priority for capturing objects of the same type, probabilities should be accordingly split among this set of rules. Let’s say, if we had three rules, each one would have a probability of one third.

In this example, the values *min* and *max* of the LHS OCU field *Prob* of the first RCU (that nearest to the GOC output) must be (resp.) set to 0 and 1/3; the second to 1/3 and 2/3, and the farthest one to 2/3 and 3/3. As stated in sub-Section 5.2, every OCU launched by the GOC had stored previously a random number (between 0 and 1 in this case), so that this random number has essentially predetermined which rule is going to capture this current OCU when it circulates along the OCB. As the RCUs are connected sequentially, this technique prevents that the first rule of the OCB have more probabilities than the rest. In this example, circulating OCUs are always captured by one of the RCUs. Moreover, if the designer wanted to provide an overall probability P_x for a rule, previous values *min* and *max* should be accordingly adjusted with respect to P_x. To sum up, the designer can easily arrange which is the probability that an object is captured by any of the RCUs. This probability setting is a programming task, external to the architecture and can be previously assigned according to the P system definition. Thanks to this treatment of probabilities, this architecture can be used to implement models of P systems using probabilities, e.g. Population Dynamics P systems (PDP systems) [5].

5.3. Global Object Container

This component is merely an OCU store, responsible of launching and receiving objects from/to the OCB. The responsible of setting up the sequence of OCUs in the OCB is the MCU, which implements the operational main loop of the membrane too.

Fig. 3 shows a scheme of the GOC. When the GOC gets an OCU from its left side (according to this figure), it writes this incoming OCU’s contents in that GOC OCU whose index is that of the field *Index* of the incoming OCU. On the other hand, when the GOC launches an OCU selected by the MCU to its right side, it copies this OCU’s fields to an outgoing OCU. Moreover, the field *Index* of this outgoing OCU is filled with the position of the GOC OCU, and the field *Random* with a new random number produced by the GOC Control Unit.

There can also be present in the GOC those special objects that modify the membrane’s behavior, like the object δ. This object would dissolve the membrane when its active multiplicity is greater than zero. Furthermore, there are three additional OCU (not being real P system objects but controlling ones), which can be launched to the OCB: those representing elements $NULL, \alpha$ and Ω. These are launched to the OCB by an MCU order, according to Algorithm 1.

Another issue that must be exposed is the generation of random numbers in the GOC. For this issue, it must be kept in mind that a pseudo-random number generator can be easily implemented with a linear-shift register as can be seen in [48] which is used to fill the *Random* field of the OCU that will be sent to the OCB.

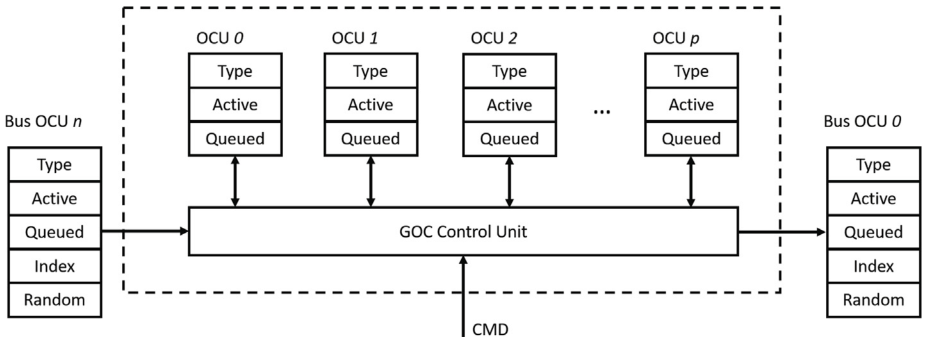

Fig. 3. Global Object Container.

Finally, in order to guarantee that the random number sequence generated by this mechanism is not always the same, a seed (or first random number) must be introduced in the generator when the system is initialized, by means of a command from the MCU.

Finally, this hardware entity allows the following operations:

- Launch a specific OCU to the OCB.
- Launch *NULL*, α or Ω to the OCB.
- Receive a OCU from the OCB, and store it at the location indicated by *index*.
- Initialize the random seed in the outgoing OCU.
- Initialize values of the inside OCUs.
- Read a value from the incoming OCU, or from an inside OCU (this is useful to output final results at the end of the computation).

5.4. Object Circulating Bus (OCB)

This hardware entity is a chain of OCUs connected between them, by means of RCUs, implementing something similar to an OCU shift register. It is a parallel bus that connects multi-bit registers (OCU) allowing multiple bits to be transferred from OCU to RCU and from RCU to OCU in a single clock cycle. OCB connects OCU and RCU in chain by means of parallel buses between its elements. There is no need to implement any Control Unit in this entity. Only a common signal *Reset* is required to reset the fields *Type* of all its OCUs to *NULL*.

5.5. Membrane Control Unit (MCU)

This is the central control unit of the system. It is responsible of:

- Allowing the initialization of the system.
- Organizing the circulation of objects through the OCB and GOC.
- Assess the completion of a computation step and the end of the computation.
- Managing the external signals.

Fig. 4 shows a diagram of the whole system with its external signals. For the sake of simplicity, internal control signals that connect the MCU with the rest of the components are not depicted. The purpose of the external signals are:

- (Input) CLK. Our architecture model is synchronous with this signal. By means of this signal the MCU pumps objects through the OCB and allows the evolution of rules in the RCUs. Because of this, an internal clock signal for GOC pumping is generated from CLK, given a fixed execution latency of the RCUs and given that once RCUs finish their operation, a pumping cycle can be produced in order to circulate objects from one rule to another.
- (Output) *REQ_END*: It indicates that the system has reached a possible end of the computation. No more rule executions were done in the last computation step.
- (Output) *REQ_CS*: It indicates that the system has reached a computation step (only for debugging purposes).
- (Input/Output) Data: Bus for transferring data, depending on the command signals.
- (Input) Command: Control bus that sends commands to the MCU, which in turn manages the different components of the system.

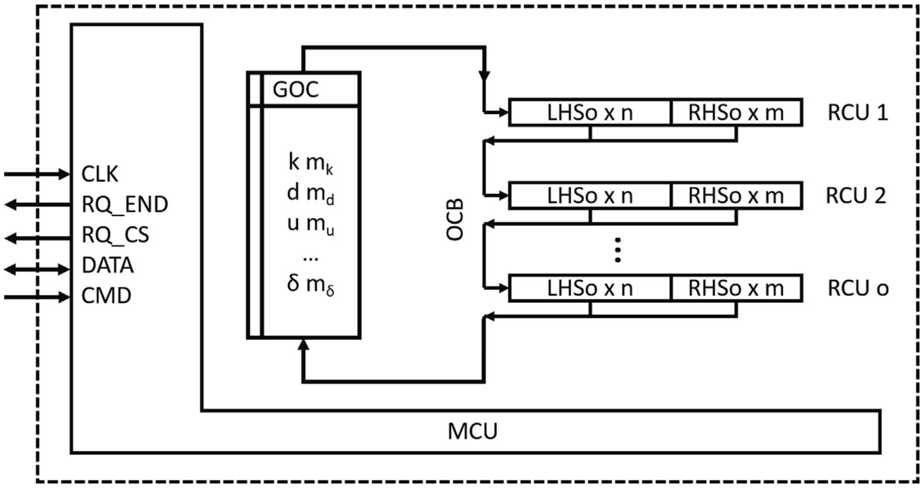

Fig. 4. A membrane with its external signals.

In Fig. 4 the whole system can be seen with its external signals. Note that although it does not appear, there is an internal control bus managed by the MCU that connects the MCU with the rest of the internal components.

Command signals must fully program the system at the beginning of the computation, that is, they can reset the MCU to its initial state, specify the objects contained in the GOC initially, fill all the RCU fields and reset the OCB to a *NULL* state.

In conclusion, it is worth to mention that our architecture model comprising these hardware components can be reprogrammed externally so that it can execute most of P systems. In this sense, our model is not an ad hoc solution for a specific P system or a sort of P systems, but a generic membrane processor. More precisely, we follow a general-purpose approach because MAREX defines a generic template for rules, in such a way that the implementation could be used for any type of rewriting rule (communication or evolution, with or without cooperation, and with or without probabilities associated to them). The only limitations that restrict our architecture are further discussed in Section 7.1 and are mainly related to practical implementation reasons as finite hardware resources.

6. MAREX-SIM: A simulator for the architecture

In order to show the operation principle of the architecture, a simulator has been developed. The simulator serves to check the architecture's operation principle, and also as a reference for testing further VHDL implementations.

It must be emphasized here that the operation principles of MAREX have been defined to emulate the synchronous processing of a potential electronic implementation of a P system. This opens the possibility of building in the short term a real universal membrane machine.

The simulator is a console application written in C# that emulates the operation of each component described above. It simulates one membrane processor which can be programmed (loading rules in its RCUs and loading objects in its GOC) and evolves at each period by means of the stimulus of a clock signal CLK, until the membrane reaches a *REQ_CS* (a computation step) or a *REQ_END* (no reaction was produced during a step). The implications of these two signals will be further discussed in section 7.4.

MAREX-SIM[1] runs only in one thread and no parallelism is implemented by now.

With respect to the random numbers of the outgoing OCUs that are going to be read by the RCUs, a standard random generator feeds the simulated GOC Control Unit. It is initialized at the start of each simulation, allowing different random sequences to be produced, depending on the initial seed.

Additional features implemented in the simulator are the following:

- Definition of the system gathering policy (common for all the RCUs): altruistic or egoistic.
- A command line interface to introduce the following parameters: model to be simulated (a few example models have been already incorporated), model size (for sizable models), random seed, and RCU object gathering policy.

The output of a simulation can be stored at different depth levels:

- Level 1: A text file containing a complete step by step simulation log.

[1] https://github.com/danicas1971/MAREX-SIM.

- Level 2: The final content of the GOC, including object types and their active multiplicities.
- Level 3: Only the number of pumping cycles elapsed at the end of the simulation.

The command line interface allows the launching of massive simulation batches, which can be subsequently compared via the gathered outputs of levels 2 and 3.

6.1. A brief simulation example

The evolution of a P system over the proposed architecture is described below. For the sake of conciseness, only the first computation step of the P system evolution is completely detailed. The subsequent computation steps work in a similar manner. The P system (see the definition in P system 1) is composed by three altruistic rules of the same priority and three initial instances of objects A, B, C (each one with an active multiplicity of one). Rules r_1 and r_2 compete for object A while objects B, C and D are not subjected to rule competition.

P system example.

Initial membrane structure: $[]_1$
Initial multiset for membrane labeled 1: A, B, C
$r_1 \equiv [AB \to C]_1$
$r_2 \equiv [AC \to BD]_1$
$r_3 \equiv [D \to E]_1$

Since this P system contains competition between rules r_1 and r_2 for the object A, proper probabilities for gathering this object must be assigned to the left side of these RCUs.

Table 1 shows the probabilities assigned to each one, which are 50% for rule r_1 and 50% for rule r_2. In spite that OCUs go sequentially across the OCB, r_1 and r_2 would compete for A with the same probability. In order to do this, see sub-Section 5.2, the field *Prob* of OCU A in r_1 contains (min, max) = [1,50] and the same field in OCU A of r_2 contains [51, 100]. When an object A is launched by the GOC, it is given a random number between 1 and 100; this implies that if r_1 did not gather A, it would pass to r_2, which finally would capture A (with the same probability as r_1).

The rest of probabilities are simply 100% because no further competition exists.

When P system rules and their characteristics are loaded and MAREX-SIM is started, a complete description of the architecture system contents is displayed (along numerous cycles) using the following format. OCUs contained in the GOC are displayed in the first line, indicating type, the separator symbol *, active multiplicity and queued multiplicity. For instance, A*1/0 means that GOC contains objects of type A with an active multiplicity of 1 and a queued multiplicity of 0. The same for objects B, C, D, E.

```
GOC = A*1/0 B*1/0 C*1/0 D*0/0 E*0/0
(α*0/0) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0
```

Under the GOC contents a row displays each ongoing OCU (between parentheses) that composes the OCB and its associated RCU content (on the right). The character in brackets denotes the rule status, which can be: active (*) or purge (P).

For those LHS and RHS OCUs residing in RCUs, the same notation than OCB OCUs is used. However, the meaning of the symbols for RCUs are: "type * needed/ active" for LHS OCUs, and "type * produces/ queued" for RHS OCUs.

In cycle 1 (first pumping cycle) $\alpha * 0/0$ has been pumped to the first rule. Rule r_1 clears its *purge* flag and makes no action on its adjacent OCU. Thus, this OCU is copied "as it is" to the following entry in the OCB (left of the second rule) in the next cycle.

```
GOC = A*1/0 B*1/0 C*1/0 D*0/0 E*0/0
(A*0/0) [*] A*1/1 B*1/0 → C*1/0
(α*0/0) [*] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0
```

Table 1
P system example. Object probabilities for each rule.

Rule	A	B	C	D
r_1	50%	100%		
r_2	50%		100%	
r_3				100%

In cycles 2 and 3, the same actions are performed on α by the following rules r_2 and r_3 as before. However, in cycle 2, A*1/0 is captured by r_1, which now contains A*1/1 (see the first LHS OCU). r_1 does not still have enough objects to be executed; so, it does not produce any new object at the right side. At the end of this pumping cycle, r_1 modifies the content of the outgoing OCU, showing A*0/0 as the result of the rule computation. This OCU is passed to r_2 in the next pumping cycle.

GOC = A*1/0 B*1/0 C*1/0 D*0/0 *E**0/0

(B*0/0) [*] A*1/0 B*1/0 → C*1/1
(A*0/0) [*] A*1/0 C*1/0 → B*1/0 D*1/0
(α*0/0) [*] D*1/0 → *E**1/0

During cycle 3 B*1/0 is captured by r_1, which produces one instance of C at its right side. In consequence, r_1 erases the active multiplicity of its LHS OCUs by the quantities indicated in the field "needed" of these OCUs (one for A and one for B respectively). Note that no object can be captured by rule r_2 because the active multiplicity of its ongoing OCU is zero (A*0/0).

In cycle 4 C*1/0 arrives at r_1, which purges its right side, and writes C*1/1 on this outgoing OCB OCU. Rules r_2 and r_3 do not produce any modification on their outgoing OCB OCUs.

GOC = A*1/0 B*1/0 C*1/0 D*0/0 *E**0/0

(C*1/1) [*] A*1/0 B*1/0 → C*1/0
(B*0/0) [*] A*1/0 C*1/0 → B*1/0 D*1/0
(A*0/0) [*] D*1/0 → *E**1/0

In cycle 5 A*0/0 enters in the GOC changing its value. Then, the GOC control unit notices that this object multiplicity has been modified and, therefore, a computation step is not issued. Besides, C*1/1 arrives at r_2, which captures its multiplicity, thus modifying the active multiplicity of the outgoing OCB OCU to C*0/1.

GOC = A*0/0 B*1/0 C*1/0 D*0/0 *E**0/0

(D*0/0) [*] A*1/0 B*1/0 → C*1/0
(C*0/1) [*] A*1/0 C*1/1 → B*1/0 D*1/0
(B*0/0) [*] D*1/0 → *E**1/0

GOC = A*0/0 B*0/0 C*1/0 D*0/0 *E**0/0

(*E**0/0) [*] A*1/0 B*1/0 → C*1/0
(D*0/0) [*] A*1/0 C*1/1 → B*1/0 D*1/0
(C*0/1) [*] D*1/0 → *E**1/0

Objects go passing through the OCB on the same way until cycle 7, where Ω*0/0 arrives at r_1, indicating the end of the object burst. As r_1 has not captured any object in its LHS, it does not set its *Purge* flag.

GOC = A*0/0 B*0/0 C*0/1 D*0/0 *E**0/0

(Ω*0/0) [*] A*1/0 B*1/0 → C*1/0
(*E**0/0) [*] A*1/0 C*1/1 → B*1/0 D*1/0
(D*0/0) [*] D*1/0 → *E**1/0

In cycle 8 Ω passes to r_2 and sets its *Purge* flag. As rule r_2 has objects in its left side to be purged, it increases the Ω's active multiplicity by one, in order to stress this circumstance to the GOC control unit, which in turn prevents the computation step by this manner. Besides, GOC CU starts filling with NULL objects the OCB until Ω enters again in the OCB. This way, the processor prevents the processing of more objects until the computation step can be assessed.

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(Ω*1/0) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(E*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(Ω*1/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

In cycle 11 objects pass again through the OCB. Nevertheless, this time all of them have no active multiplicity so rules can not capture them. Now α does not pass again, avoiding the reset of the RCUs. The only possible action is to purge objects from left or right sides in order to turn them back to the GOC. An example of this can be seen in cycle 14, where C is purged from the left side of r_2, giving as result C*1/1 in the r_2's outgoing OCB RCU. When object C arrives at the GOC again (see cycle 16), its CU would realize that the value of this object has changed; so the computation step will not be completed when Ω will arrive at the GOC in cycle 19.

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(A*0/0) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(B*0/0) [*] A*1/0 B*1/0 → C*1/0
(A*0/0) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(C*0/1) [*] A*1/0 B*1/0 → C*1/0
(B*0/0) [*P] A*1/0 C*1/1 → B*1/0 D*1/0
(A*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(D*0/0) [*] A*1/0 B*1/0 → C*1/0
(C*1/1) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(B*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*0/1 D*0/0 E*0/0
(E*0/0) [*] A*1/0 B*1/0 → C*1/0
(D*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(C*1/1) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(Ω*0/0) [*] A*1/0 B*1/0 → C*1/0
(E*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(D*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(Ω*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(E*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(Ω*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

In cycle 20 a new object burst starts. This time, there are no objects to capture (there is no active multiplicity in any object of the GOC). Besides, there are no queued or active multiplicities to be purged in the rules. Thus, when Ω passes through the OCB, its active multiplicity would not be increased allowing a new computation step in cycle 28, because there has not been any change in the GOC objects.

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(A*0/0) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(B*0/0) [*] A*1/0 B*1/0 → C*1/0
(A*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(C*1/1) [*] A*1/0 B*1/0 → C*1/0
(B*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(A*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(D*0/0) [*] A*1/0 B*1/0 → C*1/0
(C*1/1) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(B*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(E*0/0) [*] A*1/0 B*1/0 → C*1/0
(D*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(C*1/1) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(Ω*0/0) [*] A*1/0 B*1/0 → C*1/0
(E*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(D*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(Ω*0/0) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(E*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(Ω*0/0) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*1/1 D*0/0 E*0/0
(NUL) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

In cycle 28, a computation step ends. This implies that the queued multiplicities of all GOC objects are added to their respective active multiplicities and zeroed, being ready for the next computation step. This can be seen in cycle 29, where C changes to C*2/0 and the rest of queued multiplicities are already zero.

GOC = A*0/0 B*0/0 C*2/0 D*0/0 E*0/0
(α*0/0) [*] A*1/0 B*1/0 → C*1/0
(NUL) [*P] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

GOC = A*0/0 B*0/0 C*2/0 D*0/0 E*0/0
(A*0/0) [*] A*1/0 B*1/0 → C*1/0
(α*0/0) [*] A*1/0 C*1/0 → B*1/0 D*1/0
(NUL) [*] D*1/0 → E*1/0

Finally, it must be stressed that α resets r_2's *Purge* flag when it passes by its side, as it happens with the rest of rules, allowing to capture objects again.

6.2. Simulation results

Once the operation principle and the architecture were explained, performance of two very different P systems is here evaluated: a confluent and a non confluent system. The P system 2 generates the first N Fibonacci numbers. It is a deterministic P system, i.e., it is a P system with only one possible branch in its computational tree. Therefore, the system is (trivially) confluent, i.e., each possible computation generates exactly the same output. The working alphabet is $a_i : 1 \leqslant i \leqslant N, a_{i,0} : 1 \leqslant i \leqslant N+1$ and $a_{i,1} : 2 \leqslant i \leqslant N+2$. The system halts when there are no rules that can be selected for execution. This happens exactly after $N+1$ steps of computation. When the system halts, the output of the system is given by the multiplicities of objects $a_i : 1 \leqslant i \leqslant N$, and such multiplicities represent the first N Fibonacci numbers, i.e., $a_1 = 1, a_2 = 1, a_3 = 2, a_4 = 3, a_5 = 5$, etc. Objects $a_{i,0} : 1 \leqslant i \leqslant N+1$ and $a_{i,1} : 2 \leqslant i \leqslant N+2$ are used for intermediate computations in non-final configuration steps. There are four rule patterns as enumerated in P system 2. Rule patterns r_1 and r_2 produce the Fibonacci numbers. Rule patterns r_3 and r_4 are used for erasing residual objects $a_{N+1,1}$ and $a_{N+2,0}$ in the final configuration.

P system definition for a Fibonacci series generator.
Initial membrane structure: $[]_1$
Initial multiset for membrane labeled $1 : a_{1,1}, a_{2,0}$
$r_1 \equiv [a_{i,1} \rightarrow a_i, a_{i+2,0}]_1 : 1 \leqslant i \leqslant N$
$r_2 \equiv [a_{i+1,0} \rightarrow a_{i+1,1}, a_{i+2,0}]_1 : 1 \leqslant i \leqslant N$
$r_3 \equiv [a_{N+1,1} \rightarrow \lambda]_1$
$r_4 \equiv [a_{N+2,0} \rightarrow \lambda]_1$

The P system 3 generates a random undirected graph with N nodes in a non-deterministic manner. It is non-confluent, i.e., different outputs can be generated by different computations starting from the same initial configuration. The system halts when there are no rules that can be selected for execution. This happens exactly after n steps of computation.

The working alphabet is $a_i : 1 \leqslant i \leqslant N, e^*_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant N$ and $e_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant n$. Objects $a_i : 1 \leqslant i \leqslant N$ represent nodes in the graph. Objects $e_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant N$ represent edges in the graph. Objects $e^*_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant N$ are used for intermediate calculus and represent all possible edges in the graph. There are two rule patterns r_1 and r_2 competing for the same objects $a_i, a_j, e^*_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant N$. For each combination, only one rule pattern (r_1 or r_2) is selected in a non-deterministic manner. Both rule patterns consume the corresponding possible edge $e^*_{i,j}$, but only the first rule pattern produces an edge $e_{i,j}$ in the graph. Henceforth, in the final configuration there would not be objects $e^*_{i,j}$ and there would be an arbitrary set of objects $e_{i,j}$.

P system definition for a Random Graph generator.

Initial membrane structure: $[]_1$
Initial multiset for membrane labeled 1:
$a_i : 1 \leqslant i \leqslant N$
$e^*_{i,j} : i < j \leqslant N, 1 \leqslant i \leqslant N$
{Rules r_1 and r_2 generate a random graph in a non-deterministic manner}
$r_1 \equiv [a_i, a_j, e^*_{i,j} \rightarrow a_i, a_j, e_{i,j}]_1 : i < j \leqslant N, 1 \leqslant i \leqslant N;$
$r_2 \equiv [a_i, a_j, e^*_{i,j} \rightarrow a_i, a_j]_1 : i < j \leqslant N, 1 \leqslant i \leqslant N;$

To gather performance results, the stochastic nature of a P system has been always considered. In the first case, Fibonacci is deterministic (i.e. different executions starting from the same initial conditions will always lead to the same steps and the same solution), so only one simulation experiment has been done for each N. In the second case, the model is not confluent; thus 200 simulation experiments were done by initializing the architecture with a different random seed each time. Execution cycles were calculated as the average of 200 trials for each N. Table 2 shows such timing results and also, the standard deviation and typical error for every N. The difference in execution times between trials with the same N is mainly due to the operation principle (see Section 4). Recall that objects are circulated to the rule units until all rules are fed or there are no more objects left. The number of iterations in such a loop depends on the order in which objects are pumped out, as well as the number of objects gathered by each rule. Such values are different for each iteration with the aim of simulating the non-deterministic behavior of P systems.

Another aspect to be kept in mind in the simulations is the rule's gathering policy. Results were gathered for both altruistic and egoistic policies in the two models. It must be observed that multiplicity of objects is always 0 or 1 in the Random Graph model, thus, taking "all available objects" (egoistic) is exactly the same behavior as taking "only one" (altruistic). Therefore, only one curve is represented in Fig. 6. On the other hand, in the Fibonacci model multiplicity of objects can be 0 or n ($n \in Z^+, n \geqslant 1$). In this case, each behavior corresponds to a different curve in Fig. 5,6.

Finally, in both models simulation ends when no more rules can be executed and therefore, the system cannot further evolve. This condition is assessed when the architecture raises the signal *REQ_END*.

6.3. FPGA implementation

For the hardware implementation, the proposed P system architecture has been synthesized for a Xilinx Ultra96 evaluation platform. This device is a Multiprocessor System on Chip (MPSoC) which hosts application ARM processors in the processing system (PS) part and a FPGA in the programmable logic (PL). The MAREX configuration software runs in PS while the processor is deployed in the PL. Table 3 shows the parameters and FPGA resources occupancy of two versions of the MAREX architecture. As a proof of concept, a lite version (MAREX Lite[2]) has been first synthesized for the P system shown in Section 6.1, being its behavior the same as the one obtained in the software MAREX simulator. This synthesis uses a 50 MHz clock and the computation of the P system takes 820 clock cycles, resulting a real computation time of 16.4 µs. In addition, a second synthesis was done to determine the maximum processor size which can be synthesised on the specified FPGA platform (MAREX Ultra). A deeper study of possible implementations and computing time for several P systems is intended for future works. It is important to point out that the final states of the GOC are exactly equal for both implementations than those given by MAREX simulator.

7. Discussion and future work

7.1. Design's limitations

Obviously, a main concern of a hardware-oriented design like this is about the size of the P system that can be programmed within it. On this sense, the number of OCUs implementable in a specific silicon wafer is limited. There must be a limit to the maximum number of OCUs in the GOC, number of rules, and components within the rule LHSs and RHSs. Therefore, given a maximum number of implementable OCUs, the designer must choose between an architecture with very few but long rules or another one with a massive number of small rules. In addition, in order to ease the implementation of a general purpose system, all the rules should have the same structure and number of OCUs in its left-hand and right-hand

[2] https://github.com/jarios86/marex

Table 2
Performance of Random Graph model in clock cycles.

N	Average	Std. Dev.	Std. Error
2	111.19	75.14	5.313
3	2191.53	1313.30	92.864
4	15040.27	7104.80	502.385
5	65565.10	27004.13	1909.480
6	199865.70	76155.99	5385.042
7	534122.98	204873.88	14486.771
8	1246267.05	442453.86	31286.212
9	2549992.02	782009.13	55296.396

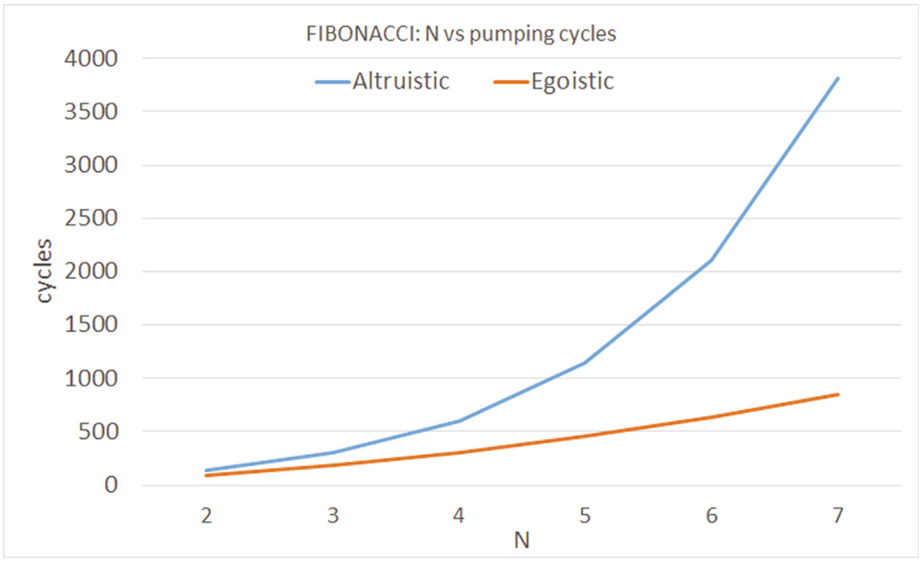

Fig. 5. Performance of Fibonacci model for several values of *N*.

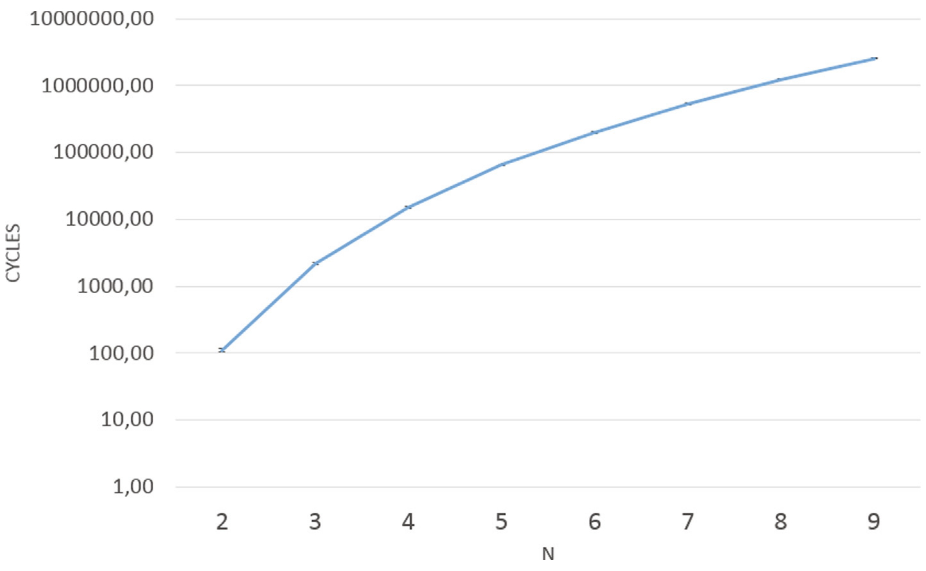

Fig. 6. Performance of Random Graph model for several values of *N*. Vertical lines denote standard error..

Table 3
MAREX version parameters for FPGA implementations and hardware resources.

	MAREX Lite	MAREX Ultra
Object types	12	64
Num Objects per type	16	32
RCUs	8	40
RCU Size	4 − 2	16 − 8
LUTs	4.91%	88.95%
Registers	1.33%	27.05%
BlockRam	0	0
DSPs	0	0

sides. This implies a limitation over the P systems to be programmed, that can be overcome making different versions of the architecture with different sizes.

In this work we present the first implementation of this architecture and for simplicity we have considered the case of systems composed of a single membrane. Nevertheless, as shown in [2] a transition P system having multiple membranes can be emulated by another transition P system with only one membrane. Therefore, our architecture has a universal scope (in Turing sense), i.e. it can be used to implement any transition P system, simply applying a normalization process to the definition of the P system. In addition, in future works we will study the possible extension of the proposed architecture to include multiple membranes.

This can be useful to increase the parallelism degree of the implementation, depending on the semantics of the system.

With respect to performance issues, we are aware that the main bottleneck of the architecture is the OCB and the manner in which OCUs are passed through it.

In the current version objects are presented one by one to the rules and gathered by them; and this process is repeated again and again until no rule execution can be performed. In fact, this circumstance produces (together with the purge mechanism and the probability of gathering objects) repeated bursts of objects circulating through the OCB that produce no rule executions and therefore degrade system's performance. Thus, parallel computation occurs only when several OCUs in different RCUs react at the same time (see Section 7.3 for a deeper discussion).

This performance degradation can be partially overcome with a wider bus in which more than one OCU is presented to each rule. This would allow a more efficient computation but would require more hardware resources, which can limit the number of rules for certain designs. Future works will evaluate this compromise between increasing rule execution parallelism and spending more hardware resources.

Finally, we must bear in mind the competitive nature of rules, which is implemented in our architecture using random policies for gathering objects. This aspect, together with the dependency of rules on the order in which objects are launched from the GOC, does have an influence in the evolution of the P system. Thus, a P system in our architecture that has achieved an early solution (signaled by setting up the *REQ_END* signal) might have reached another solution if the system had continued evolving.

For this reason, *REQ_END* signal activation does not imply that the architecture has found the unique definitive final state, but that the P system cannot evolve more with the chain of events that happened up to that moment (see Section 7.4 for further discussion). In fact, a different OCU order may allow the P system to continue evolving, and thus to reach another *REQ_END* with a different final state.

7.2. Executing non-deterministic computations

As stated in previous sections, one of the main concerns about this architecture is its dependency on the object launching order for arriving to different possible solutions when we have rules competing for objects (thus depending on certain "gathering probabilities").

In order to evaluate such circumstance, a conceptual test has been made in MAREX-SIM: the model Random Graph for $N = 4$ has been executed using 2720 different random seeds, for two types of initial object GOC ordering: the order on which the objects are described in the system specification (called here "primary order") versus a random order depending on the random seed placed in the membrane processor.

For this system the maximum number of edges for a non-directed graph with $n = 4$ vertexes is $n(n-1)/2 = 6$, henceforth there are $2^6 = 64$ possible different computations, i.e., 64 non-directed different graphs that can be generated with 4 vertexes.

Note that we have chosen a little value for N in this experiment because this ensures a small number of possible solutions, exactly, 64. By means of this value, we pursue two objectives: a small execution time (the complete set was executed in less than 10 min) and, illustrating bar charts of manageable sizes. In contrast, if we have just used $N = 5$, there would have been more than 1.04×10^6 possible solutions, and, consequently, the probability bar chart would have been hard to visualize.

Next is to register graph frequencies, that is, how many times each graph appears in the set of executions. Fig. 7 shows that, even though the machine reaches all the possible solutions, frequencies are different for primary order machine and for random ordering machine. Therefore, it is concluded that the order on which the objects are launched from the GOC have an influence on the solution to be reached.

In order to make a less biased machine, we tested some additional GOC arrangements trying to make frequencies more homogeneous:

- Initial random ordering.
- Object rotation after Ω has entered in the GOC.
- Reverse ordering after Ω has entered three times in the GOC

It must be taken in mind that these order arrangements were implemented in MAREX-SIM only for testing if solutions are biased by them, but they are not part of the core implementation. In the simulator, after a computational step the GOC can be reordered according to one of the arrangements commented before.

Fig. 8 depicts the same experiment with previous three arrangements. It is seen that frequency distributions are different but biasing is not reduced at all. We can conclude that biasing of final results for such a stochastic P system is inherent to it,

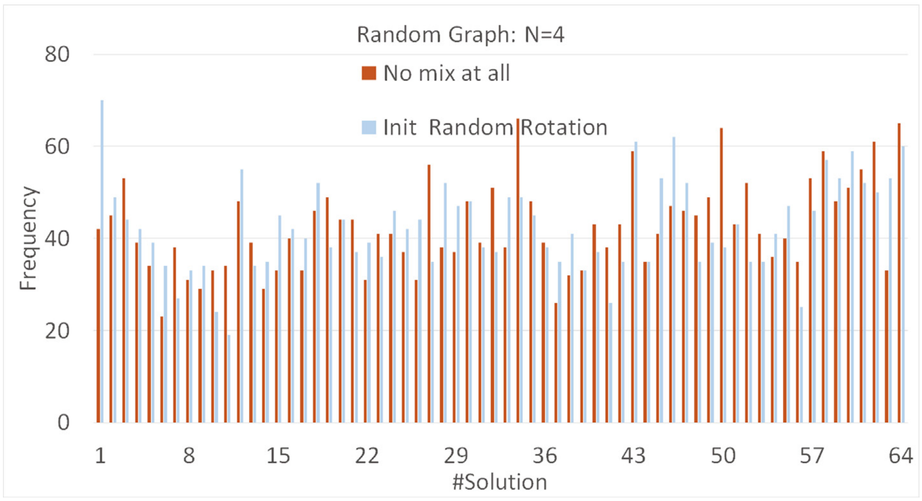

Fig. 7. Testing dependency on object launching order.

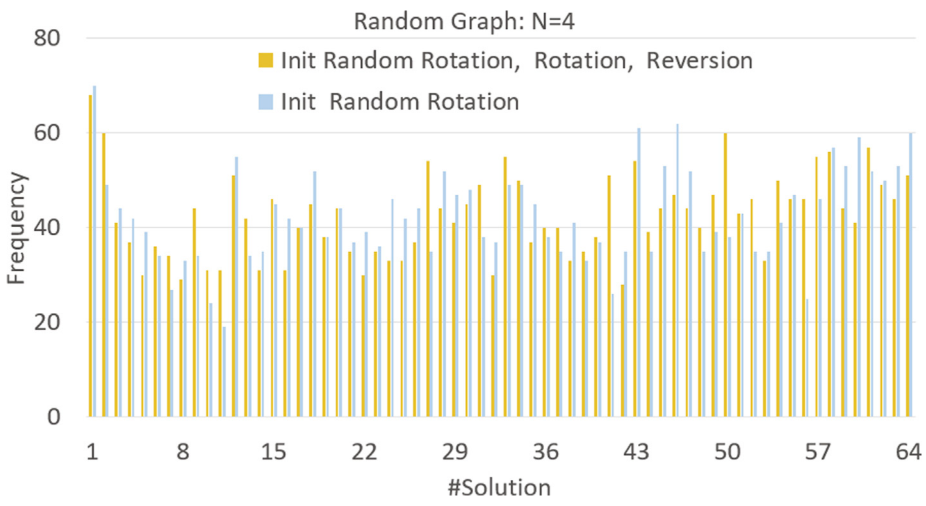

Fig. 8. Evaluating bias with object mixing.

and it depends on many factors like random seed, GOC object ordering, RCU ordering. Some additional tests were done using different configurations of these factors, but all of them conclude that the graph distribution is always far from a nearly homogeneous one.

7.3. Exploiting architecture's parallelism

As seen in previous sections, every RCU can work independently, being capable of collecting objects from the OCB, creating new (RHS) objects and sending them to the GOC (via the OCB). Although RCUs are chained to the OCB and objects are shifted one by one through this OCB, this sequential shifting does not prevent parallel rule execution at every cycle. That is to say, there may be more than one rule execution at one cycle. The exact number of rules per cycle depends not only on the P system specification, but also on the sequence in which the objects are circulating through the OCB. The more objects are presented at the same time to those RCUs that need them, the more rules would probably be executed in the same cycle.

Specifically, previously exposed P systems, which were introduced to demonstrate the architecture's operation, did not take much advantage of the potential architecture's parallelism. Let us define a typical performance measurement having some similarity with that of *CPI* (Cycle Per Instruction for conventional Von Neumann processors): *CPR*, that counts for the average number of Cycles Per executed Rule. Obviously, the less *CPR* a P system execution exhibits, the more parallelism it offers to our architecture.

Table 4 shows the performance of the previous exposed P systems for several configurations: Fibonacci using an altruistic mode (Fibo Alt) and Fibonacci in egoistic mode (Fibo Ego) for 6, 8, ..., 16 rules, and the Random Graph Generator (Rand G) for different numbers of rules.

For each configuration, the *Burst*, which is the number of cycles that a complete object burst needs to pass through the OCB, the number of objects, mean number of cycles (mean number of cycles used to simulate the P system, making 200 experiments), number of rules executed, and the *CPR* are shown.

As expected for the Rand G P system, its *CPR* grows exponentially with the size of the system (N), because the bigger N is, the more objects are circulating in the OCB, and therefore, the less probable is that the needed set of objects arrive in a burst to the proper rule.

In the case of Fibonacci, things are different because a rule just needs an object to be executed, and each rule is to be executed many times in a single computation step.

In the egoistic mode (Fibo Ego in Table 4), one rule gets all the active multiplicity of one object all at once. Then, if there were objects with an active multiplicity greater than one, the rule would be executed at the rate of one time per cycle until the rule consumes all the multiplicity of this object in its left side. As a result, *CPR* would remain approximately constant independently of the size of the system.

Conversely, if this P system were run in altruistic mode (Fibo Alt in Table 4), each rule would just get the necessary active object multiplicity to be executed once. Then, the remaining objects would be dropped to the GOC again, so, they would arrive to the same rule in the next object burst of the GOC (needing *Burst* cycles to be assigned again). This would continuously occur until these objects have been completely consumed. As expected, altruistic mode results in a higher *CPR* than the egoistic one, but with a growth rate approximately linear with the system size.

Table 4
Performance table for two P systems: Fibonacci model using both altruistic (Fibo Alt) and egoistic mode (Fibo Ego), and Random Graph model.

	N	#Rules	Burst	#Objs	Cycles	Ex	*CPR*
Fibo Alt	2	6	15	8	138.00	12	11.50
	3	8	20	11	304.00	20	15.20
	4	10	25	14	605.00	33	18.33
	5	12	30	17	1146.00	54	21.22
	6	14	35	20	2107.00	88	23.94
	7	16	40	23	3808.00	143	26.63
Fibo Ego	2	6	15	8	93.00	12	7.75
	3	8	20	11	184.00	20	9.20
	4	10	25	14	305.00	33	9.24
	5	12	30	17	456.00	54	8.44
	6	14	35	20	637.00	88	7.24
	7	16	40	23	848.00	143	5.93
Rand G	2	2	7	4	111.19	1	111.19
	3	6	16	9	2191.53	3	730.51
	4	12	29	16	15040.27	6	2506.71
	5	20	46	25	65535.10	10	6556.51
	6	30	67	36	199865.70	15	13324.38
	7	42	92	49	534122.98	21	25434.43
	8	56	121	64	1246267.05	28	44509.54
	9	72	154	81	2549992.02	36	70833.11

Previous examples give us an idea of the parallelism available in a P system depending on its rule definition and the configuration mode that a rule uses to capture objects from the GOC. However, could a case be made with an almost optimal use of the parallelism for our architecture proposal? In order to develop this case, P system 4 has been built and tested.

P system for testing parallelism.

Initial membrane structure: $[]_1$
Initial multiset for membrane labeled 1:
$a_i \; * n : 1 \leqslant i \leqslant n$
{Rule r_1 consumes any a_i to produce b_i}
$r_1 \equiv [a_i \rightarrow b_i] : 1 \leqslant i \leqslant n;$

In P system 4 the number of objects and rules grows linearly as n increases (recall that $a_i \; * n$ represents n copies of object a_i).

For the total P system completion, the number of computation steps needed are simply two: a first one to drain all the a_i objects, and a second one to raise the *REQ_END* signal. In addition, the number of executed rules is equal to the square of n, because there are n rules, each one processing a set of n objects at the same time.

If egoistic mode were used for this P system 4, every rule would get all the active object multiplicity at once. As objects a_i have active multiplicity n, once a rule gets all the object multiplicity, it would be executed n times lasting one cycle per execution.

In the optimal case, the first rule would catch its corresponding set of objects at cycle 1 and it would begin to execute at that cycle. Then, at the cycle 2, the first rule would be executed a second time while the second rule would get its set of objects, being simultaneously executed for the first time. Correspondingly, at cycle 3, rules 1 and 2 would react once more, while the third rule would catch its objects and execute once, and so on. Hence, several rules can be executed in parallel at every cycle in this system.

This explains how *CPR* decreases to zero as n increases (see Table 5), and also, how the total number of cycles grows linearly even when the number of rule executions grows in a quadratic fashion. Therefore, we can conclude that the degree of parallel rule execution for our architecture model depends on the specification of a P system.

More exactly, it depends on the design of the rule competitiveness and the number of elements available for them. For example, if there were rules that depend on the objects generated by other RHS rules (e.g. $r_1 = ab \rightarrow c, r_2 = cd \rightarrow f$, where r_2 can only be executed after the r_1 has been completed), the parallel execution of the dependent rules would not be possible until a computational step upgrades the active multiplicity of these objects. This circumstance also occurs in conventional computers where the code's efficiency depends on the real dependencies of its data (e.g. $x = a + b; y = x + w$, where x and y cannot be calculated in parallel).

7.4. Future research

In its current scheme, our hardware model is fully compliant with a basic transition P system defined in one membrane. Nonetheless, future extensions to multi-membrane systems can be straightforwardly accomplished by interconnecting those membranes through a set of buses.

These membranes would be activated, deactivated and related to other membranes during the execution cycles of the whole P system, maybe by means of a central controller, which would be also in charge of updating the membrane hierarchies.The design of more scalable versions of our architecture model and their implementation in FPGAs are foreseeable future research fields.

In this respect, implementation of division and mitosis can also be extensions of a multi-membrane architecture, because dividing a membrane implies the incorporation (at runtime) of a new processor with its proper connections to the already

Table 5
Performance table for the P system 4 using egoistic mode.

n	#Rules	Burst	#Objs	Cycles	Ex	CPR
10	10	31	20	126	100	1.260
50	50	151	100	606	2500	0.242
100	100	301	200	1206	10000	0.121
150	150	451	300	1806	22,500	0.080
200	200	601	400	2406	40,000	0.060
250	250	751	500	3006	62,500	0.048
300	300	901	600	3606	90,000	0.040
350	350	1051	700	4206	122,500	0.034
400	400	1201	800	4806	160,000	0.030

existing membranes. In this case, decisions must be made about how to include on the fly new processors and interconnections next to an existing one.

In the same line, building policies to make more flexible architectures to accomplish large membranes in more than one circuitry unit (e.g. FPGA) would need improving the functionality of the MCU and some kind of sharing circuitry yet to be researched.

Another feature needed to be implemented in a multi-membrane architecture is the dissolution. As it can be seen in Section 4, the dissolution object has been included in this version of the architecture in order to be part of a future multi-membrane version. It must be said that two main changes must be addressed to implement dissolution. First, a new output signal REQ_{DIS} must be added to the architecture in order to indicate the dissolution state. Second, the central controller must collect this signal from every membrane to execute the dissolution just in case. When a dissolution is going to be executed (at the end of a computational step), the central controller must transfer the objects from the dissolved membrane to its parent, and also, must disconnect the dissolved membrane from the system. For this reason, the central controller must be able to interact with membranes through its programming bus to perform all the above commented actions.

Another issue related with the dissolution is the fact that it cannot be known in advance how many objects are involved in a dissolution rule, and, consequently, how many objects are to be added to the parent membrane. This can be partially solved in a static way if all P system GOCs contain all the possible objects that can be transferred between membranes (that is, the union of all P system objects). Thus, after a dissolution the parent's GOC would result the sum of itself and the GOC of the dissolved membrane.

Apart from deploying our architecture in physical devices, extended monitoring tools for debugging purposes will be needed to check the correct evolution of the emulated P systems.

Another improvement for this architecture is the extension to probabilistic P systems, where the rule depends on a probability to be executed once it has captured all its necessary objects. The probabilistic mechanism used by the OCUs to collect objects from the OCB may be a solution, along with some new fields in the OCB and a new random number generator.

Another issue that must be addressed for any P system execution is that of the halting assertion. There is always the possibility that a P system contains a halting condition difficult to be reached, or even that there is no halt condition at all. Namely, this can happen if there is a rule that is competing with one or more other rules for an elevated number of objects (see sub-Section 7.2) or if the rule is conditioned by a very low probabilistic term in a Population dynamics P system (PDP system) [5].

Although the designer should be conscious of this problem, a P system machine implementation may incorporate mechanisms to detect such difficult halting. On one hand, simple warning mechanisms, like detecting when the system has not evolved after a specific number of times, can be smoothly inserted in our architecture.

On the other hand, more precise strategies may imply changes in the current architecture and have to be elaborated in future research. For instance, the introduction of special checking modes in each RCU that detects the possibility of reactions by recording any circulating OCB object (but without modifying the active multiplicity of the OCB objects, so as to allow the rest of RCUs to do the same checking) is a plausible solution for next versions of our architecture.

Finally, and in order to explore the full potential of this new architecture, new compilation and synthesis tools to go from a P system (e.g. specified in P-lingua [28]) to an implementation compliant with the proposed architecture are also interesting functionalities to develop in the near future.

8. Conclusions

The first intrinsically parallel hardware architecture (MAREX) for executing generic P systems in the form of a membrane processor is introduced in this work. Its main building blocks have been described, and its viability and correctness have been demonstrated by means of a simulator that reproduces the architecture's operating principles.

Through the simulated execution of several P systems, it is unveiled that "programming" our architecture to execute any transition P system is feasible and remarkably beneficial. Specifically, our universal architecture has been checked: a) to demonstrate that a transition P system reaches all its possible solutions emerging from the object competitiveness among rules. b) to evaluate its performance when the complexity of a P system grows in an exponential fashion. c) to implement different object gathering policies.

The architecture proposal operates as a set of independent functional units connected through a daisy chain bus, allowing the parallel execution of several rules. Thus, a proper design of a P system would take advantage of this performance benefit if it contained a great number of independent rules (without common objects in their left-hand side).

Although this foundational architecture proposal has been introduced in its simplest form, i.e. like a one-membrane processor, it can be easily extended to a multi-membrane architecture by chaining several of these processors in a daisy chain fashion while laying on the same operation principles. However, daisy chain architectures are not parallel and further investigation to look for better parallel multi-membrane architectures are needed. Thus, further developments point to a complete and scalable multi-membrane architecture covering the two intrinsic levels of parallelism: inside a membrane and among several membranes. It is also interesting to study the potentials that P system hardware could offer to interact with other systems, for instance, in order to control, supervise or predict their behavior.

This work opens new research lines, specially promising for the practical resolution of problems where P systems have already demonstrated their potential.

CRediT authorship contribution statement

Daniel Cascado-Caballero: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Fernando Diaz-del-Rio:** Conceptualization, Methodology, Validation, Formal analysis, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration. **Daniel Cagigas-Muñiz:** Conceptualization, Methodology, Validation, Writing - original draft, Writing - review & editing, Visualization. **Antonio Rios-Navarro:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing. **Jose-Luis Guisado-Lizar:** Validation, Writing - original draft, Writing - review & editing, Visualization. **Ignacio Pérez-Hurtado:** Methodology, Validation, Resources, Writing - original draft, Writing - review & editing. **Agustín Riscos-Núñez:** Resources, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by the Ministerio de Ciencia e Innovacion of Spain and the AEI/FEDER (EU) through the research projects TIN2017-89842-P (MABICAP), PID2019-110455GB-I00 (Par-HoT).

References

[1] G. Păun, Computing with membranes, J. Comput. Syst. Sci. 61 (1) (2000) 108–143, https://doi.org/10.1006/jcss.1999.1693.
[2] G. Păun, Membrane Computing: An Introduction, Springer-Verlag, Berlin, Germany, 2002.
[3] M.J. Pérez-Jiménez, F. Sancho-Caparrini, A formalization of transition P systems, Fundamenta Informaticae 49 (1–3) (2002) 261–271, special Issue: Membrane Computing (WMC-CdeA2001) Guest Editor(s): Carlos Martín-Vide, Gheorghe Păun..
[4] G. Păun, P systems with active membranes: Attacking NP-Complete problems, Journal of Automata, Languages and Combinatorics 6 (1) (2001) 75–90, and CDMTCS TR 102, Univ. of Auckland, 1999 (www.cs. auckland.ac.nz/CDMTCS)..
[5] M.A. Colomer, A. Margalida, M.J. Pérez-Jiménez, Population dynamics P system (PDP) models: a standardized protocol for describing and applying novel bio-inspired computing tools, PLOS One 8 (4) (2013) 1–13, https://doi.org/10.1371/journal.pone.0060698, url:https://doi.org/10.1371/journal. pone.0060698.
[6] G. Zhang, M. Gheorghe, L. Pan, M.J. Pérez-Jiménez, Evolutionary membrane computing: a comprehensive survey and new results, Inf. Sci. 279 (2014) 528–551, https://doi.org/10.1016/j.ins.2014.04.007, url:http://www.sciencedirect.com/science/article/pii/S002002551400454X.
[7] G. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez, Tissue P systems with cell division, Int. J. Comput. Commun. Control 3 (3) (2008) 295–303, https://doi.org/10.15837/ijccc.2008.3.2397, url:http://univagora.ro/jour/index.php/ijccc/article/view/2397.
[8] B. Song, X. Zeng, A. Rodríguez-Patón, Monodirectional tissue P systems with channel states, Inf. Sci. 546 (2021) 206–219, https://doi.org/10.1016/j.ins.2020.08.030, url:http://www.sciencedirect.com/science/article/pii/S0020025520307970.
[9] M. Ionescu, G. Păun, T. Yokomori, Spiking neural P systems, Fundamenta Informaticae 71 (2,3) (2006) 279–308.
[10] L. Pan, G. Păun, G. Zhang, F. Neri, Spiking Neural P systems with communication on request, Int. J. Neural Syst. 27 (8) (2017) 1750042. doi:10.1142/S0129065717500423..
[11] T. Wu, F. Bible, A. Paun, L. Pan, F. Neri, Simplified and yet turing universal Spiking Neural P systems with communication on request, Int. J. Neural Syst. 28 (8) (2018) 1850013. doi:10.1142/S0129065718500132..
[12] D. Orellana-Martín, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, A path to computational efficiency through membrane computing, Theoret. Comput. Sci. 777 (2019) 443–453, https://doi.org/10.1016/j.tcs.2018.12.024.
[13] A. Leporati, L. Manzoni, G. Mauri, A.E. Porreca, C. Zandron, A survey on space complexity of P systems with active membranes, Int. J. Adv. Eng. Sci. Appl. Math. 10 (3) (2018) 221–229, https://doi.org/10.1007/s12572-018-0227-8.
[14] P. Sosík, L. Cienciala, Computational power of cell separation in tissue P systems, Inf. Sci. 279 (2014) 805–815, https://doi.org/10.1016/j.ins.2014.04.031, url:http://www.sciencedirect.com/science/article/pii/S0020025514004782.
[15] G. Păun, F.J. Romero-Campero, Membrane Computing as a modeling framework. Cellular systems case studies, in: M. Bernardo, P. Degano, G. Zavattaro (Eds.), Formal Methods for Computational Systems Biology, Vol. 5016 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 168–214. doi:10.1007/978-3-540-68894-5_6..
[16] M. Gheorghe, N. Krasnogor, M. Camara, P systems applications to systems biology, Biosystems 91 (3) (2008) 435–437, https://doi.org/10.1016/j.biosystems.2007.07.002.
[17] G. Păun, R. Păun, Membrane computing and economics: numerical P systems, Fundamenta Informaticae 73 (1,2) (2006) 213–227.
[18] I. Pérez-Hurtado, M. Martínez-del Amor, G. Zhang, F. Neri, M. Pérez-Jiménez, A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning, Integr. Comput.-Aided Eng. 27 (2) (2020) 121–138. doi:10.3233/ICA-190616..
[19] C. Buiu, C. Vasile, O. Arsene, Development of membrane controllers for mobile robots, Inf. Sci. 187 (2012) 33–51, https://doi.org/10.1016/j.ins.2011.10.007, url:http://www.sciencedirect.com/science/article/pii/S0020025511005421.
[20] S. Pang, T. Ding, X. Mao, N.N. Xiong, Design and analysis of a decision intelligent system based on enzymatic numerical technology, Inf. Sci. 547 (2021) 450–469, https://doi.org/10.1016/j.ins.2020.07.033, url:http://www.sciencedirect.com/science/article/pii/S0020025520306952.
[21] H. Peng, J. Wang, M.J. Pérez-Jiménez, A. Riscos-Nóóez, An unsupervised learning algorithm for membrane computing, Inf. Sci. 304 (2015) 80–91, https://doi.org/10.1016/j.ins.2015.01.019, url:http://www.sciencedirect.com/science/article/pii/S0020025515000572.
[22] D. Díaz-Pernil, H.A. Christinal, M.A. Gutiérrez-Naranjo, Solving the 3-col problem by using tissue P systems without environment and proteins on cells, Inf. Sci. 430–431 (2018) 240–246, https://doi.org/10.1016/j.ins.2017.11.022, url:http://www.sciencedirect.com/science/article/pii/S0020025516313305.

[23] A. Yan, H. Shao, Z. Guo, Weight optimization for case-based reasoning using membrane computing, Inf. Sci. 287 (2014) 109–120, https://doi.org/10.1016/j.ins.2014.07.043, url:http://www.sciencedirect.com/science/article/pii/S002002551400752X.
[24] G. Zhang, M.J. Pérez-Jiménez, M. Gheorghe, Real-life Applications with Membrane Computing, Vol. 25, Springer, 2017. doi:10.1007/978-3-319-55989-6..
[25] D. Díaz-Pernil, C. Graciani-Díaz, M.A. Gutiérrez-Naranjo, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Software for P systems, Ch. 17, Oxford University Press, 2010, pp. 437–454.
[26] L. Valencia-Cabrera, D. Orellana-Martín, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, An interactive timeline of simulators in membrane computing, J. Membr. Comput. 1 (3) (2019) 209–222, https://doi.org/10.1007/s41965-019-00016-z.
[27] D. Díaz-Pernil, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, A P-Lingua programming environment for membrane computing, in: D.W. Corne, P. Frisco, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 187–203.
[28] I. Pérez-Hurtado, D. Orellana-Martín, G. Zhang, M.J. Pérez-Jiménez, P-lingua in two steps: flexibility and efficiency, J. Membr. Comput. 1 (2019) 93–102, https://doi.org/10.1007/s41965-019-00014-1.
[29] M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Simulating P systems on GPU devices: a survey, Fundamenta Informaticae 136 (3) (2015) 269–284, https://doi.org/10.3233/FI-2015-1157.
[30] M. Á. Martínez-del-Amor, D. Orellana-Martín, I. Pérez-Hurtado, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Design of Specific P Systems Simulators on GPUs, in: T. Hinze, G. Rozenberg, A. Salomaa, C. Zandron (Eds.), Membrane Computing, Vol. 11399 of Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 202–207. doi:10.1007/978-3-030-12797-8_14..
[31] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulation of P systems with active membranes on CUDA, Brief. Bioinf. 11 (3) (2010) 313–322, https://doi.org/10.1093/bib/bbp064.
[32] G. Zhang, Z. Shang, S. Verlan, M.A. Martínez-del Amor, C. Yuan, L. Valencia-Cabrera, M.J. Pérez-Jiménez, An overview of hardware implementation of membrane computing models, ACM Comput. Surveys 53 (4) (2020) 1–38. doi:10.1145/3402456..
[33] G. Zhang, M.J. Pérez-Jiménez, A. Riscos-Núñez, S. Verlan, S. Konur, T. Hinze, M. Gheorghe, Membrane Computing Models: Implementations, Springer (2021), https://doi.org/10.1007/978-981-16-1566-5.
[34] B. Petreska, C. Teuscher, A reconfigurable hardware membrane system, in: C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 269–285.
[35] C. Teuscher, From membranes to systems: Self-configuration and self-replication in membrane systems, Biosystems 87 (2) (2007) 101–110, papers presented at the Sixth International Workshop on Information Processing in Cells and Tissues, York, UK, 2005. doi:10.1016/j.biosystems.2006.09.002. url:http://www.sciencedirect.com/science/article/pii/S0303264706001547..
[36] S. Alonso Villaverde, L. Fernández Muñoz, F. Arroyo Montoro, F.J. Gil Rubio, A circuit implementing massive parallelism in transition P systems, Int. J. Inf. Technol. Knowl. 2 (1) (2008) 35–42. url:http://oa.upm.es/2194/..
[37] S. Alonso, L. Fernandez, F. Arroyo, F.J. Gil, Main modules design for a hw implementation of massive parallelism in transitl1233ion P-systems, Artif. Life Robot. 13 (2008) 107–111, https://doi.org/10.1007/s10015-008-0526-4.
[38] S.M.G. Canaval, A.G. Rodriguez, S.A. Villaverde, Hardware implementation of P systems using microcontrollers. An operating environment for implementing a partially parallel distributed architecture, in: 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2008, pp. 489–495.
[39] A. Gutiérrez, L. Fernández, F. Arroyo, S. Alonso, Hardware and software architecture for implementing membrane systems: a case of study to transition P systems, in: M.H. Garzon, H. Yan (Eds.), DNA Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 211–220.
[40] V. Martínez, A. Gutiérrez, L.F. de Mingo, Circuit FPGA for active rules selection in a transition P system region, in: M. Koppen, N. Kasabov, G. Coghill (Eds.), Advances in Neuro-Information Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 893–900.
[41] V. Nguyen, D. Kearney, G. Gioiosa, An implementation of membrane computing using reconfigurable hardware, Comput. Inf. 27 (2008) 551–569.
[42] V. Nguyen, D. Kearney, G. Gioiosa, An algorithm for non-deterministic object distribution in P systems and its implementation in hardware, in: D.W. Corne, P. Frisco, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 325–354.
[43] V. Nguyen, D. Kearney, G. Gioiosa, A region-oriented hardware implementation for membrane computing applications, in: G. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 385–409.
[44] V. Nguyen, D. Kearney, G. Gioiosa, An extensible, maintainable and elegant approach to hardware source code generation in reconfig-P, J. Logic Algebr. Program. 79 (6) (2010) 383–396, membrane computing and programming. doi:10.1016/j.jlap.2010.03.013. url:http://www.sciencedirect.com/science/article/pii/S1567832610000299.
[45] J. Carnero, D. Díaz-Pernil, H. Molina-Abril, P. Real, Image segmentation inspired by cellular models using hardware programming, Image-A 1 (3) (2010) 142–150.
[46] D. Kulakovskis, D. Navakauskas, Automation of metabolic P system implementation in FPGA: A case study, in: 2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), 2015, pp. 1–4..
[47] J. Quiros, S. Verlan, J. Viejo, A. Millan, M. Bellido, Fast hardware implementations of static P systems, Comput. Inf. 35 (2016) 687–718.
[48] T.G. Birdsall, M.P. Ristenbatt, Introduction to linear shift-register generated sequences, Tech. rep., The University of Michigan, 1958..