

A Reconfigurable Hardware Membrane System

Biljana Petreska and Christof Teuscher

Logic Systems Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL),
CH-1015 Lausanne, Switzerland
E-mails: biljana.petreska@epfl.ch, christof@teuscher.ch
URL: <http://lslwww.epfl.ch>

Abstract P systems are massively parallel systems and software simulations do not usually allow to exploit this parallelism. We present a parallel hardware implementation of a special class of membrane systems. The implementation is based on a universal membrane hardware component that allows to efficiently run membrane systems on specialized hardware such as FPGAs. The implementation is presented in detail as well as performance results and an example.

1 Introduction

For more than half a century, the von Neumann computing architecture (i.e., the stored program concept)—first been expressed in 1945 [25]—largely dominates computer architecture in many variants and refinements. Even the much-heralded parallel machines of various designs are just collections of von Neumann machines that possess a communication structure superimposed on the underlying machine. Of course, some alternative approaches were developed, however, they always occupied a marginal place, mainly due to their limited applicability. One might certainly ask if the von Neumann architecture is a paradigm of modern computing science, what is its future? Nobody can anticipate the future, however, there seems to exist a growing need for alternative computational paradigms and several indicators support this observation.

The 21st century promises to be the century of bio- and nanotechnology. New materials and technologies [8,13] such as self-assembling systems, organic electronics, hybrid electronic-biological machines, etc., and the ever-increasing complexity and miniaturization of actual systems [1] will require to fundamentally rethink the way we build computers, the way we organize, train, and program computers, and the way we interact with computers.

A good example for such a new paradigm is *Membrane Computing*, which might be considered as part of the general intellectual adventure of searching for new progress in computer science. Membranes are crucial in Nature: they define compartments, they ensure that substances stay inside or outside it (separator), they let certain molecules pass through it (filter), and they form a communication structure. The membrane forms a boundary between the cell (the “self”) and its environment and is responsible for the controlled entry and exit of ions.

The autonomous organization and self-maintenance of this boundary is an indispensable feature for living cells. This is also the basic idea behind Varela and Maturana's *autopoietic* systems [23].

Membrane Computing (MC) or *P systems*, initiated by Paun in 1998 [14], is a highly parallel—though theoretical—computational model afar inspired by biochemistry and by some of the basic features of biological membranes. Whereas Paun's membrane computing amalgamates in an elegant way membranes and artificial chemistries, various other systems with membranes exist. For example, Langton's self-replicating loops [10] make use of a kind of membrane (i.e., a state of the CA's cells) that encloses the program. The embryonic projects, on the other hand, uses cellular membranes to divide the empty space into a multicellular organism [12,20]. Explicit membranes (i.e., membranes with a material consistence) are not always required: the POETic¹ project [22], for example, is based on a hierarchical organization of molecules and cells with implicit separations.

The main goal of the work presented in this paper was to develop a hardware-friendly computational architecture for a certain class of membrane systems. The question of whether to simulate systems in software or to implement them in (specialized) hardware is not a new one (see for example [9]). With the advent of *Field Programmable Gate Arrays (FPGAs)* [21,24], however, this question somehow took a back seat since it suddenly became easy and inexpensive to rapidly build (or rather "configure") specialized hardware. Currently, P systems are usually implemented and simulated on a standard computer using an existing (such as the Iasi P systems simulator [4], or one of the recently proposed distributed software simulators [5,18]) or a custom simulator. As Paun states, "[i]t is important to underline the fact that "implementing" a membrane system on an existing electronic computer cannot be a real implementation, it is merely a simulation. As long as we do not have genuinely parallel hardware on which the parallelism [...] of membrane systems could be realized, what we obtain cannot be more than simulations, thus losing the main, good features of membrane systems" [15, p. 379]. To the best of our knowledge, a first possible hardware implementation has been mentioned in [11].

The hardware-based implementation presented in this paper is a parallel implementation that allows to run a certain class of P systems in a highly efficient manner. The current design has been simulated and synthesized for FPGAs only, it could however very easily be used to implement an application specific integrated circuit (ASIC).

The reminder of the paper is as follows: Section 2 provides an description of the modified P system we used. Section 3 describes the implementation in detail. The Java application as well as a typical design flow are presented in Section 4. Section 5 presents a simple example and the performance results obtained. Finally, Section 6 concludes the paper.

¹ See also: <http://www.poeticissue.org>.

2 Description of the Membrane System Used

A classical P system² (see [15,16] for a comprehensive introduction) consists of cell-like membranes placed inside a unique “skin” membrane (see Figure 1). Multisets of *objects*—usually multisets of symbols-objects—and a set of *evolution rules* are then placed inside the regions delimited by the membranes. Each object can be transformed into other objects, can pass through a membrane, or can dissolve or create membranes. The evolution between system configurations is done nondeterministically by applying the rules synchronously in a maximum parallel manner for all objects able to evolve. A sequence of transitions is called a *computation*. A computation *halts* when a halting configuration is reached, i.e., when no rule can be applied in any region. A computation is considered as successful if and only if it halts.

P systems are not intended to faithfully model the functioning of biological membranes, rather they form a sort of abstract *artificial chemistry (AC)*: “An artificial chemistry is a man-made system which is similar to a real chemical system” [6]. ACs are a very general formulation of abstract systems of objects which follow arbitrary rules of interaction. They basically consists of a set of molecules S , a set of rules R , and a definition of the reactor algorithm A . By abstracting from the complex molecular interactions in Nature, it becomes possible to investigate how the AC’s elements change, replicate, maintain themselves, and how new components are created.

To be able to efficiently implement P systems in hardware, we had to modify classical P systems in the following two points:

1. The rules are *not* applied in a maximum parallel manner but following a predefined order.
2. The P system is *deterministic*, i.e., for a given initial configuration, the systems always halts in the same halting configuration.

The main reason is that a straightforward implementation of classical P systems would have been too expensive in terms of hardware resources needed. We were basically interested by a minimal hardware architecture and not primarily by a faithful classical P systems implementation. More details on the reactor algorithm will be given in Section 3.4.

3 Description of the Implementation

In this section, the hardware implementation of the membrane system shall be described in detail. The implementation basically supports cooperative P systems with priority using membrane dissolution and creation. Note that the source code of the implementation and further information is freely available on the following web-site: <http://www.teuscher.ch/psystems>. The resulting design is a

² Numerous variations exist meanwhile. The P systems web-site provides further information: <http://psystems.disco.unimib.it>.

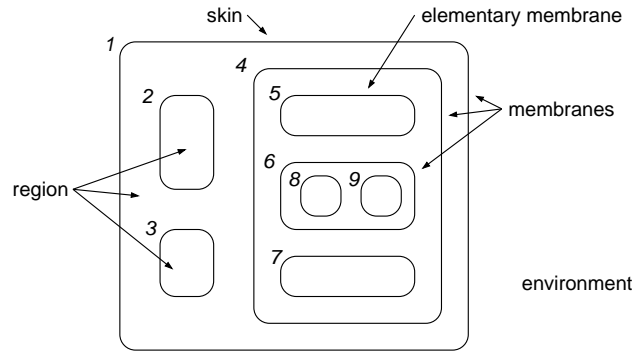


Figure 1. Elements of a membrane system represented as a Venn diagram. Redrawn from [15].

universal membrane module that can be instantiated and used anywhere in a membrane system.

For the current implementation, we have chosen the FPGA technology as it is an ideal platform for prototyping hardware systems, is fully reprogrammable, and offers very good performance at a low cost. A *Field Programmable Gate Array* (FPGA) circuit [17,21,26] is an array of (a usually large number of) logic cells placed in a highly configurable infrastructure of connections. Each logic cell can be programmed to realize a certain function (see also [21] for more details).

In addition, once a design has been implemented on an FPGA, it can rather easily be transferred to a full custom *Application Specific Integrated Circuit* (ASIC) technology, which usually provides even better performance.

3.1 Membrane Structure

The membranes of our implementation are without a material consistence, i.e., they do not exist as physical frontiers. Therefore, when referring to the membrane, we will actually refer to its *contents*, i.e., to the multisets of objects and the evolution rules of the region it encloses (see Figure 2). The membrane’s main functionality of physically separating two regions can be completely and unambiguously replaced by representing the relations between the membrane’s contents (see Figure 3, A). According to Paun’s membrane definition, the size and the spatial placement (distances, positions with respect to any coordinates) do not matter [15, p. 51] and we therefore decided to dispose the membranes in the two-dimensional space in an arbitrary manner. The only information that matters and that allows to preserve the membrane structure are the relations (i.e., the interconnections and the hierarchical organization) between the cellular membranes (see Figure 3, B).

The relations between the membranes—which are represented in our implementation by electronic buses (a collection of wires)—are basically only used when objects are being transferred between two membranes. A membrane can

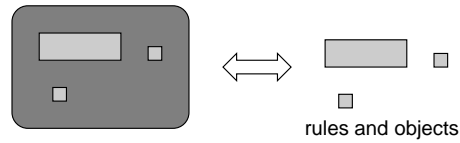


Figure 2. Membranes are not explicitly represented, instead, the cell is defined by its contents.

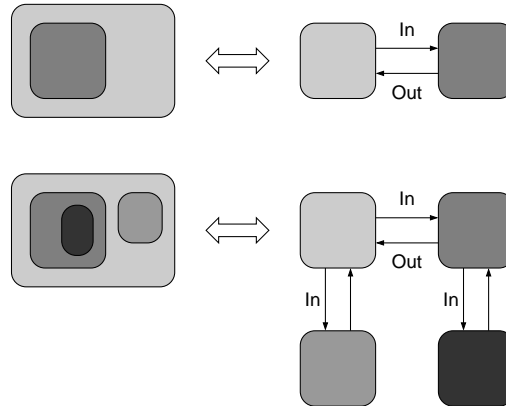


Figure 3. The relations between the membranes are represented by the interconnections.

send and receive objects only from its external enclosing membrane and from the membranes it contains on the next lower level. For example: membrane 4 of Figure 1 can send and receive objects from its external membrane 1 (upper-immediate) and from 5, 6, and 7 (lower-immediate). However, membrane 4 cannot directly communicate with membranes 8 and 9 nor with 2 and 3. Therefore, connections in the form of bi-directional communication channels must exist between a membrane, its lower-immediate and upper-immediate membranes (see Figure 5, A). In the following, the two communication directions shall be described.

First, one communication channel (i.e., an electronic bus) is required to *send* objects to the lower-immediate membranes (see Figure 5, B). The bus originates from the upper membrane (1) and all the lower membranes are connected to it. It is possible that several lower-immediate membranes are connected to the same bus (e.g., membranes 2 and 3). To directly address a given message to a specified membrane, a label is attached to each message that is sent on the bus. Each membrane compares the label of each message with its own label and accordingly decides whether it is the receiver or not.

Second, to *receive* objects from the lower-immediate membranes, one bus for each lower-immediate membrane would basically be required as objects can be received simultaneously (Figure 5, C). We did not adopt this solution as it

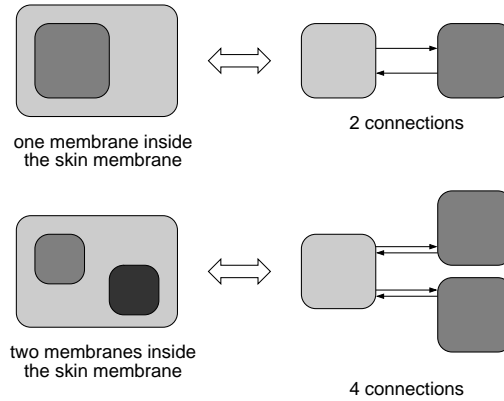


Figure 4. The number of connections basically depends on the membrane structure, which represents a serious problem for hardware implementations. Our implementations avoids this problem by using a special bus.

is dependent on the membrane structure. To avoid multiple buses, we simply replaced them by a bus that first “traverses” all the lower-immediate membranes (i.e., membranes 2 and 3) before it gets connected to the target membrane (1). Each lower-immediate membrane basically adds its objects to the message that is transferred on the bus before passing it on to the next membrane in the chain.

Naturally, a membrane can be at the same time an upper-immediate (if it contains other membranes) and a lower-immediate for another membrane (membrane 2 of Figure 5, D, for example). Therefore, each membrane must basically possess the connections required to connect to both of these levels (Figure 5, E and F), although some of them might not always be used.

Our architecture is completely independent of the surrounding membrane structure. For example, as shown in Figure 4, the number of interconnections is basically dependent on the number of the lower-immediate membranes. Each time a new membrane is being created, a new bus would have been required. Our final membrane implementation—as shown in Figure 5, F—is a sort of universal component which avoids this serious drawback and which might therefore be used anywhere in the hierarchy of a membrane system.

3.2 Multisets of Symbol-Objects

Each region of a membrane can potentially host an unlimited number of *objects*, represented by the symbols from a given alphabet. In our case, these objects are not implemented individually but only their *multiset* is represented. The multiplicity of each set, i.e., number of identical objects, is stored within a register³ as shown in Figure 6. The register’s order reflects the order of the objects within the

³ A memory element with rapid access.

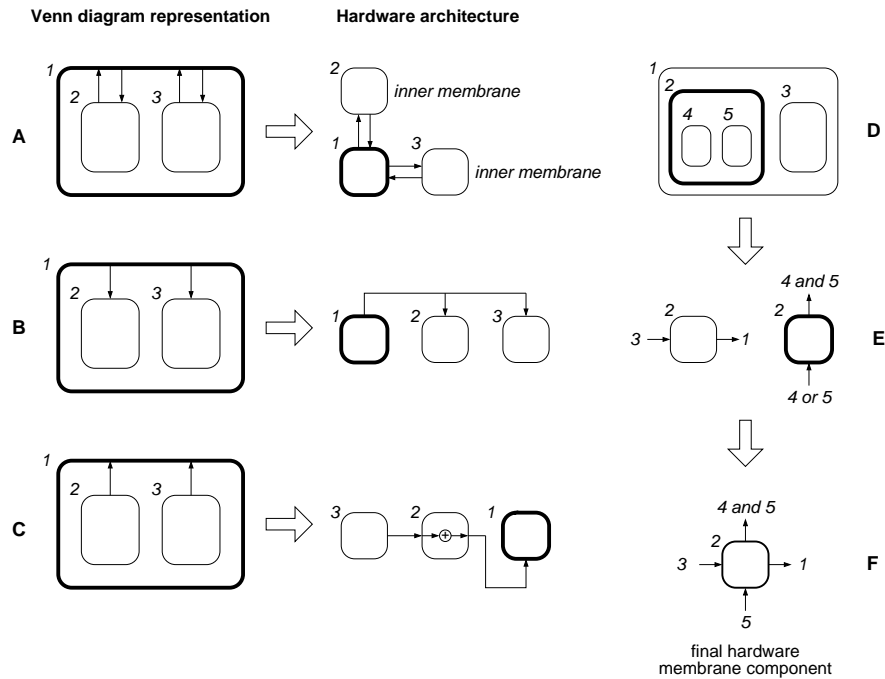


Figure 5. Possible connection structures between membranes. Sub-figure F shows the final membrane hardware component with its interconnection structure. The membrane component is universal and might be used anywhere in the hierarchy of a membrane system.

alphabet and consequently, the register position directly indicates which symbol's multiplicity is being stored. Note that the size of the register limits the maximum number of object instances that can be stored. In our implementation, you can choose between 8-bit and 16-bit registers, which allows to store up to $2^7 = 128$ or up to $2^{15} = 32768$ instances of an object (the remaining bit is being used to detect a capacity overflow).

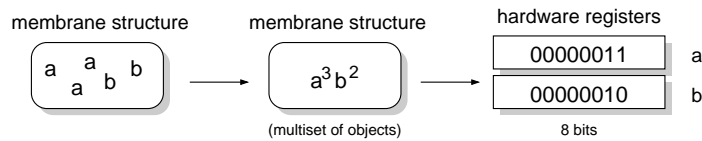


Figure 6. Representation of the symbol-objects in hardware. Alphabet = {a, b}.

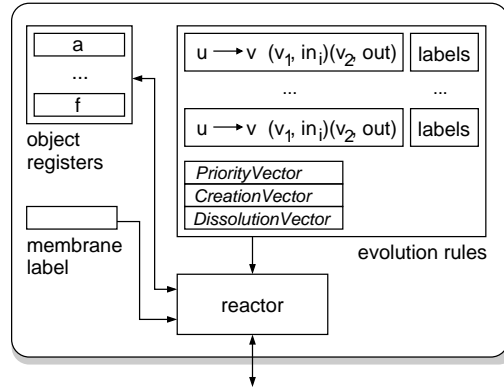


Figure 7. A view within the membrane hardware component.

3.3 Evolution Rules

An evolution rule is a rewriting rule of the form $u \rightarrow v(v_1, in_i)(v_2, out)$ that replaces the membrane objects u by the objects v , sends the objects v_1 to the lower-immediate membrane with label i , and sends the objects v_2 to the upper-immediate membrane. *Cooperative* rules, i.e., rules with a length of u greater than 1, are also supported in our implementation. There is only one possible *in target* command in our implementation, whereas in some P systems there may be several. However this simplification does not significantly affect the computational power and could easily be modified in a future extension of the design.

The module responsible for the evolution rules (see also Figure 7) is composed of the following elements (for more details on the priority and the membrane dissolution and creation, see also Sections 3.5 and 3.6):

- 4 arrays of 8-bit registers (4 multisets of objects) store the *left-hand side*, *here*, *out* and *in* parts of a rule (i.e., u , v , v_1 , and v_2 in the formula);
- one 8-bit register contains the *in target* label (i.e., i from in_i);
- 8-bit registers store up to 4 labels of rules with higher priority;
- one 8-bit register specifies the label of the membrane to be created (in case of a creation rule);
- one bit in the 8-bit *PriorityVector* register is set to “1” if the rule is priority-dependant;
- one bit in the 8-bit *DissolutionVector* register indicates whether the rule will dissolve the membrane; and
- one bit in the 8-bit *CreationVector* register indicates whether the rule will create a membrane.

All these registers are initialized only at the beginning of the simulation and do not change afterwards. Remember also that the P system’s evolution rules do not change throughout a computation, except for dissolving membranes, where the associated set of rules disappears.

In addition, a further module constantly computes whether a rule can be applied or not. The module takes the membrane objects as inputs and generates the signal *Applicable* for each rule:

$$\textit{Applicable}(u \rightarrow v) = 1 \quad \text{iff} \quad u \leq w \quad (1)$$

where w is the multiplicity of membrane objects and $u > 0$. For example, the rule $a^3 \rightarrow b$ can only be applied if there are at least 3 objects of a present within the current membrane. Connecting all the *Applicable* signals of all the rules present in the membrane together by means of a logical or-gate then directly provides the signal indicating when the membrane's computation is finished, i.e., when all rules (that can be applied) have been applied.

3.4 Reactor Algorithm

The evolution of a P system can basically be decomposed into *micro-* and *macro-steps*. We used a somehow different decomposition as the micro- and macro-steps proposed by Paun in [14]. Our micro-step corresponds to the application of a rule inside a membrane. A macro-step then consists in applying sequential micro-steps in parallel to all membranes, as long as a rule can be applied somewhere. At the end of a macro-step (i.e., no further rule is applicable), the newly obtained and the stored objects are consolidated to the already existing objects (in all membranes in parallel). The system knows when all micro-steps have been completed due to the global signal generated on the basis of the membrane's *Applicable* signals (see Section 3.3). Algorithms 1 and 2 illustrate the details of the micro- and macro-step in our hardware implementation. Note that the rule r is of the form $u \rightarrow v(v_1, in_i)(v_2, out)$ and the “store” operation simply updates the internal registers.

Algorithm 1 Micro-step of our P system

```

Select a rule  $r$ 
if Applicable( $r$ ) then
  Remove left-hand side objects  $u$  from membrane objects
  Store right-hand side objects  $v$  in the UpdateBuffer
  Send  $v_1$  to the lower-immediate membranes (via the connection bus)
  Store  $v_2$  in the ToUpperBuffer
end if
Store objects from upper-immediate membrane in the FromUpperBuffer

```

The selection of the rules (first step of Algorithm 1) is done in the following way: a special memory inside the membrane contains an ordered list of rule labels (specified at the P system initialization). This sequence might either be generated randomly—which simulates in a certain sense a nondeterministic behavior—or in a pre-specified order. However, it is important to note that two equal initializations result in the same P system behavior, i.e., the system

Algorithm 2 Macro-step of our P system

In all membranes simultaneously
while Exists an applicable rule in the system **do**
 Perform a micro-step
end while
Send *ToUpperBuffer* objects to the upper-immediate membrane
Add *UpdateBuffer* objects to membrane objects
Add *FromUpperBuffer* objects to membrane objects
Add objects from the lower-immediate membrane to membrane objects

is completely deterministic, which is different from the original P systems idea (see also Section 2). Furthermore, the rules of a P system are usually applied in a maximum parallel manner. As Algorithm 1 illustrates, our P system hardware implementation does not allow a maximum parallel application of the rules within a micro-step. The main reason for this was that such an algorithm would have been too expensive to implement in terms of resources used as it would require to implement a search algorithm. However, the parallelism on the membrane level is fully realized and membranes compute simultaneously.

3.5 Priorities

A rule $r_1 : u_1 \rightarrow v_1$ from a set R_i is used in a transition step with respect to Π (the membrane system) if there is no rule $r_2 : u_2 \rightarrow v_2$ which can be applied at the same step and $r_2 > r_1$ [15, p. 70] (i.e., r_2 has a higher priority than r_1). Remember that there is competition for the rule application and not for the objects to be used. In order to respect the priorities, the priority-applicability of a rule is computed only once, i.e., at the beginning of each macro-step. The signal *PriorityApplicable* is generated according to the following formula:

$$PriorityApplicable(r_i) = \neg \left(\sum_j Applicable(r_j) \mid r_j > r_i \right) \quad (2)$$

The information $r_j > r_i$ is contained in the *labels* (see Section 3.3 and Figure 7). Once selected, a rule is applied only if it is both *Applicable* and *PriorityApplicable* (i.e., both signals are active).

3.6 Creating and Dissolving Membranes

The work presented so far did not allow to create and dissolve membranes. We therefore propose these two extensions to our membrane hardware architecture. Dissolving and creating membranes is both biologically and mathematically motivated. As Paun writes, it is a frequent phenomenon in nature that a membrane is broken, and that the reaction conditions in the region change and become those of the covering membrane, for all the contents of the previously dissolved membrane. Furthermore, membranes are also created continuously in biology,

for instance in the process of vesicle mediated transport. When a compartment becomes too large, it often happens that new membranes appear inside it. From an organizational point of view, membrane creation is mainly used as a tool for better structuring the objects of a given (natural or artificial) system. But interestingly, we observe also an increase of the computational power since the class of creating membranes can solve NP-complete problems in linear time (see [15, p. 311] for an example of linearly solving the SAT problem by means of P systems with membrane creation).

The first extension is the *membrane dissolving action* [15, p. 64]. The application of rules of the form $u \rightarrow v\delta$ in a region i is equivalent to using the rule, then dissolving the membrane i . All objects and membranes previously present in the membrane become contents of the upper-immediate membrane, while the rules of the dissolved membrane are removed.

The solution adopted for our hardware implementation simply consists in disabling the membrane by means of a membrane *Enable* signal. Once a membrane dissolving rule has been used, the membrane objects are sent to the upper-immediate membrane, the membrane is reconfigured to fit the new connection configuration, and the enable signal is set to “0” to “disable” the membrane. It is important to note that the membrane (i.e., its contents) continues to physically exist. This is, among other reasons, necessary to in order not to interrupt the communication busses as presented in Section 3.1. Obviously, the drawback of this solution is that the space occupied by a “removed” membrane cannot be reused.

The second extension made to our model consists in the creation of membranes [15, p. 302]. When a rule of the form $a \rightarrow [{}_i v]_i$ is used in a region j , then the object a is replaced with a new membrane with the label i and the contents as indicated by v . The label i is important as it indicates the set of rules that applies in the new region.

The solution adopted for our hardware implementation consists in creating the potentially “new” membranes at initialization already. This is possible since all parameters needed for their creation (i.e., their objects, evolution rules, and the position in the membrane structure) are already known. At the beginning, these membranes are considered as non-operational. Once a membrane creating rule has been used, one of the already created membranes will simply be activated (until they are all used). Obviously, the number of available inactive spare membranes limits the computational power of the system.

4 Design Flow and Java-Tool for Generating VHDL Code

The membrane system has been programmed using the VHDL [3,26] hardware description language. The FPGA programming process basically requires three software tools (note that many other tools exist): ModelSim⁴ (a VHDL simula-

⁴ Model Technology: <http://www.model.com>.

tor), LeonardoSpectrum⁵ (a VHDL synthesizer), and the Xilinx Design tools⁶. Figure 8 illustrates the typical design flow.

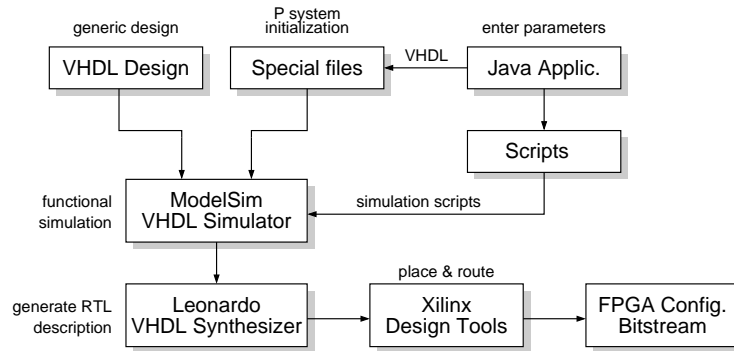


Figure 8. Illustration of the typical design flow.

The VHDL files are first compiled with the ModelSim software, which allows to simulate and debug the VHDL code on a behavioral level. Once compiled and simulated, the design is synthesized, analyzed, and optimized by means of the LeonardoSpectrum software. LeonardoSpectrum also provides all technology-relevant details with regards to the chosen hardware technology or circuit, i.e., the electrical schemas, the timing analysis, the hardware resources required, etc. In addition, LeonardoSpectrum also outputs the EDIF file which is used in the next step by the Xilinx software. The Xilinx Design tools basically map (i.e., place & route) the design on the chosen chip and generate the necessary configuration files. In our case, the software outputs the configuration bitstream for the Xilinx FPGA we have chosen as a target technology.

Since a large part of the code is dependent on the membrane system to be simulated and on its initial values, we implemented a Java application which basically automates the generation of the VHDL source code. As shown in Figure 9, the application allows to specify in a convenient way all relevant parameters of the membrane system and then automatically generates the VHDL configuration and initialization files as well as several scripts that automate the simulation process.

5 Experiments and Results

In this section, a simple educative toy example as well as a selection of results shall be presented.

⁵ Mentor Graphics: <http://www.mentor.com/leonardospectrum>.

⁶ Xilinx: <http://www.xilinx.com>.

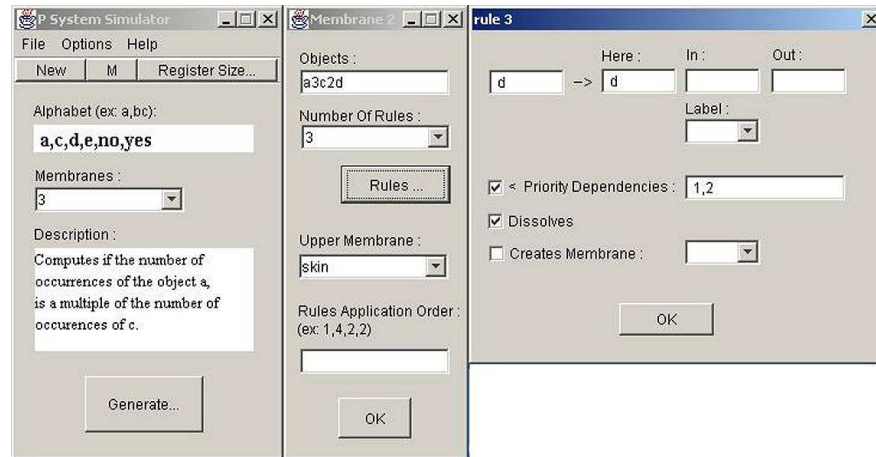


Figure 9. Java application that allows to automatically generate all necessary configuration files of the membrane system hardware implementation.

5.1 An Example

We consider the P system presented on page 73 in [15] (see also Figure 10) that allows to decide whether k divides n . Figure 11 shows parts of the timing diagram of a ModelSim simulation of this P system. The alphabet used is the following: $\text{Alphabet} = \{a, c, d, e, \text{no}, \text{yes}\}$. The signal names as shown in the simulation represent the following information:

- **clk**: Global clock signal.
- **macro_step**: Set to '1' when a macro-step is being completed and then triggers the consolidation of the objects in all membranes.
- **values i** , and **newvalues i** : Correspond to the multiplicities of objects in the membrane i . *Values i* represents the objects currently contained in membrane i . *Newvalues i* represents the objects that will appear in membrane i after a rule (in this or another membrane) has been applied.

Note that the rules are not represented in Figure 11 as they do not change during the P system evolution. A description of the simulation—which requires a total of 15 clocks steps to complete—is as follows:

1. The system is initialized. The inner membrane with label 2 contains a^3b^2d (*values2*, *init*). The system therefore decides whether $n = 3$ divides $k = 2$.
2. Rule $ac \rightarrow e$ is applied. The objects a and c in the second membrane are replaced by one object d , i.e., *values 2* has been decremented on the positions that correspond to the objects a and c (follow the alphabet order) and *newvalues 2* has been incremented at the position of the object d . Note that the rule $d \rightarrow d\delta$ cannot be applied in this macro-step as a rule of higher priority ($ac \rightarrow e$) is applicable.

3. No rules can be applied in any membrane, therefore a new macro-step is started and the *values* are updated: $values := values + newvalues$.
4. The rule $ae \rightarrow c$ is applied (the only applicable rule). Again, a macro-step and an update as in the previous step follow.
5. The rule $d \rightarrow d\delta$ is finally used as all the rules of higher priority cannot be applied (i.e., there are no more occurrences of the object a). This causes membrane 2 to dissolve in the next macro-step.
6. Beginning of a macro-step. Membrane 2 is dissolved, therefore its contents are transferred to membrane 1 (since membrane 1 contains membrane 2).
7. The rule $dce \rightarrow (no, in_3)$ is applied in membrane 1 for it has priority over the other rule. Objects dce in membrane 1 are replaced with no (**result**) in membrane 3, which is the output membrane.

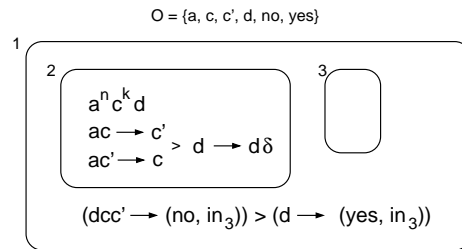


Figure 10. The membrane system simulated (page 73 in [15]).

Our implementation has been tested by means of four examples from [15] with the following features: membrane dissolution, membrane creation, transferring objects to upper- and lower-immediate membranes, and cooperative P systems with priorities. Short descriptions of the problems tested, screenshots of the ModelSim simulations, and the simulation results of the LeonardoSpectrum synthesis can be found on: <http://www.teuscher.ch/psystems>.

5.2 Results

In order to evaluate the performance of our hardware implementation, we synthesized membrane systems of different sizes and complexities. The target circuit was a Xilinx high-end Virtex-II Pro 2VP50 FPGA⁷ that roughly contains about 24000 configurable logic blocks (CLBs)⁸. Table 1 summarizes the results obtained. One can see that the resources used are directly proportional to the maximum number of objects. Furthermore, adding the possibility of membrane creation adds complexity and therefore results in a design that is almost twice as large and runs at a much slower clock speed.

⁷ <http://www.xilinx.com>.

⁸ 1 CLB = 4 logic cells, 1 logic cell = 4-input LUT + FF + Carry Logic.

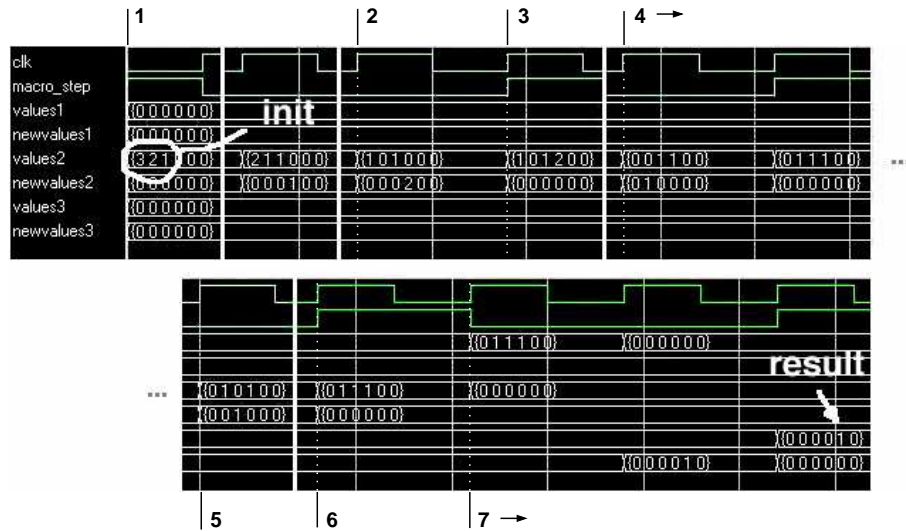


Figure 11. Screenshot of a ModelSim P system simulation. The white fat vertical lines indicate cuts on the time axis, i.e., irrelevant parts of the simulation have been removed.

At the exception of the limited physical resources of the reconfigurable circuit, i.e., the number of gates available, there is no limitation on the number of membranes, rules, and objects that might be simulated from a theoretical point of view since the universal membrane architecture is entirely parameterizable by generic parameters in the VHDL code.

6 Conclusion and Future Work

We presented a universal and massively parallel hardware implementation of a special class of P systems. The proposed hardware design is highly parameteriz-

Membranes	Objects	Resources (CLBs)	Clock frequency	Remark
10	6	1037 (4.2%)	198 MHz	dissolution only
10	12	2213 (9%)	190 MHz	dissolution only
10	6	2069 (8.4%)	45 MHz	dissolution and creation
10	12	4254 (17.3%)	31 MHz	dissolution and creation
20	6	1977 (8%)	198 MHz	dissolution only
20	12	4185 (17%)	191 MHz	dissolution only
20	6	3967 (16.1%)	45 MHz	dissolution and creation
20	12	8110 (33%)	27 MHz	dissolution and creation

Table 1. Summary of the performance results obtained for a Xilinx Virtex-II Pro 2VP50ff1517 (Speed grade: -7) FPGA.

able and can in principle be used to evolve membrane systems of any complexity as long as the underlying hardware provides sufficient resources. The architecture of the universal membrane hardware module allows to use the same module anywhere in the hierarchy of a membrane system and independently of the number of rules and objects to be stored within it. The results have shown that membrane systems can be implemented very efficiently in hardware.

Future work will concentrate on the development and improvement (in terms of speed and resources used) of the existing design. In addition, it is planned to extend the existing design in order to be able to reuse dissolved membranes and in order to apply rules in a fully parallel and nondeterministic manner. In addition, dealing with a larger number of objects would probably require a serial communication protocol as the bus size was already a serious limitation in the current implementation. The number of rules and priority dependencies (set to 8 and 4 respectively) have not yet been parameterized, but this is also envisaged in the future.

Furthermore, we also envisage to extend the current design to other important classes of P systems such as for example systems with symport/antiport [15, p. 130] and systems with membrane division [15, p. 273].

Further ongoing research consists in developing more general hardware architectures for membrane systems amalgamated with alternative computational and organizational metaphors. *Amorphous Membrane Blending (AMB)* [19], for example, is an original and not less unconventional attempt to combine some of the interesting and powerful traits of *Amorphous Computing* [2], *Membrane Computing* [15], *Artificial Chemistries* [6], and *Blending* [7]. One of the goals is to obtain a novel, minimalist, and hardware-friendly computational architecture with organizational principles inspired by biology and cognitive science.

Acknowledgments

This work was supported in part by the Swiss National Science Foundation under grant 20-63711.00, by the Leenaards Foundation, Lausanne, Switzerland, and by the Villa Reuge, Ste-Croix, Switzerland. The authors are grateful to Ludovic Righetti for his careful reading of this work and his helpful comments.

References

1. International technology roadmap for semiconductors. Semiconductor Industry Association, <http://public.itrs.net/Files/2001ITRS>, 2001.
2. H. Abelson, D. Allen, D. Coore, C. Hanson, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, May 2000.
3. P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
4. G. Ciobanu and D. Paraschiv. Membrane software. A P system simulator. *Fundamenta Informaticae*, 49(1–3):61–66, 2002.
5. G. Ciobanu and G. Wenyuan. A parallel implementation of the transition P systems. In A. Alhazov, C. Martín-Vide, and G. Paun, editors, *Proceedings of the*

- MolCoNet Workshop on Membrane Computing (WMC2003)*, volume 28/03, pages 169–169, Tarragona (Spain), 2003. Rovira i Virgili University, Research Group on Mathematical Linguistics.
6. P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries—a review. *Artificial Life*, 7(3):225–275, 2001.
 7. G. Fauconnier and M. Turner. *The Way We Think: Conceptual Blending and the Mind's Hidden Complexities*. Basic Books, 2002.
 8. S. De Franceschi and L. Kouwenhoven. Electronics and the single atom. *Nature*, 417:701–702, June 13 2002.
 9. L. Garber and D. Sims. In pursuit of hardware-software codesign. *IEEE Computer*, 31(6):12–14, June 1998.
 10. C. G. Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.
 11. M. Madhu, V. S. Murty, and K. Krithivasan. A hardware realization of P systems with carriers. Poster presentation at the Eight International Conference on DNA based Computers, Hokkaido University, Sapporo Campus, Japan, June 10–13 2002.
 12. D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Toward robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–540, April 2000.
 13. N. Mathur. Beyond the silicon roadmap. *Nature*, 419(6907):573–575, October 10 2002.
 14. G. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. First published in a TUCS Research Report, No 208, November 1998, <http://www.tucs.fi>.
 15. G. Paun. *Membrane Computing*. Springer-Verlag, Berlin, Heidelberg, Germany, 2002.
 16. G. Paun and G. Rozenberg. A guide to membrane computing. *Journal of Theoretical Computer Science*, 287(1):73–100, 2002.
 17. E. Sanchez. An introduction to digital systems. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Towards Novel Computational Architectures*, chapter 2, pages 13–47. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.
 18. A. Syropoulos, E. G. Mamatas, P. C. Allilomes, and K. T. Sotiriades. A distributed simulation of P systems. In A. Alhazov, C. Martín-Vide, and G. Paun, editors, *Proceedings of the MolCoNet Workshop on Membrane Computing (WMC2003)*, volume 28/03, pages 455–460, Tarragona (Spain), 2003. Rovira i Virgili University, Research Group on Mathematical Linguistics.
 19. C. Teuscher. *Amorphous Membrane Blending and Other Unconventional Computing Paradigms*. PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2004. To be published.
 20. C. Teuscher, D. Mange, A. Stauffer, and G. Tempesti. Bio-inspired computing tissues: Towards machines that evolve, grow, and learn. *BioSystems*, 68(2–3):235–244, February–March 2003.
 21. S. M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Boston, 1994.
 22. A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, and Alessandro E. P. Villa. Poetic tissue: An integrated architecture for bio-inspired hardware. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proceedings of the 5th International Conference (ICES2003)*, volume 2606 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2003.

23. F. Varela, H. Maturana, and R. Uribe. Autopoiesis: The organization of living systems, its characterization and a model. *BioSystems*, 5:187–196, 1974.
24. J. Villasenor and W. H. Mangione-Smith. Configurable computing. *Scientific American*, 276(6):54–59, June 1997.
25. J. von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, June 30 1945.
26. J. F. Wakerly. *Digital Design: Principles & Practices*. Prentice Hall International Inc., New Jersey, third edition, 2000.