A Hardware Membrane System

Biljana PETRESKA, Christof TEUSCHER

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

Abstract. P systems are massively parallel systems. Software simulations do no 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 expressed in 1945 [12] – 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 21^{st} century promises to be the century of bio- and nanotechnology. New materials and technologies [11, 6] such as self-assembling systems, organic electronics, hybrid electronic-biological machines, etc., and the ever-increasing complexity and miniaturization of actual systems [22] 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*, a paradigm that 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 *autopoetic* systems [19].

Membrane Computing (MC) or *P systems*, initiated by Paun in 1998 [15], 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 [8] make use of a 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 multi-cellular organism [16, 10]. Explicit membranes (i.e., membranes with a material consistence) are not always required. The POEtic[1] project [18], for example, is based on a hierarchical organization of molecules and cells with an implicit separation.

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 [7]). With the advent of *Field Programmable Gate Arrays (FPGAs)* [21], however, this question took a back seat since it suddenly became easy and inexpensive to rapidly build (or rather *configure*) specialized hardware. Currently, P systems are implemented and simulated on a standard computer using an existing (such as the Iasi P systems simulator [3]) 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" [14, p. 379]. To the best of our knowledge, a first possible hardware implementation has been mentioned in [9].

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.

2 Description of the Membrane System Used

A classical P system[2] (see [13, 14] for a comprehensive introduction) consists of cell-like membranes placed inside a unique "skin" membrane (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

[1]See also: `http://www.poetictissue.org`

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

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" [4]. 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.

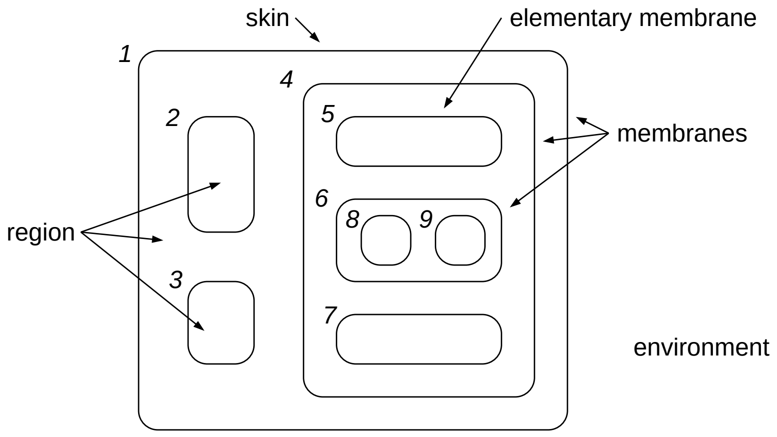


Figure 1: Elements of a membrane system represented as a Venn diagram. Redrawn from [14].

3 Description of the Implementation

In this section, the hardware implementation of the membrane system shall be described in detail. The implementation basically supports P systems with priorities and also allows to create new membranes and to dissolve existing ones. Note that the source code of the implementation and further information is freely available on the following website: `http://www.teuscher.ch/psystems`. The resulting design is a universal membrane module that can be instantiated and used anywhere in a membrane system.

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 the *contents*, i.e., multisets of objects and the evolution rules, of the region it encloses. The membrane's functionality – to physically separate two regions – can be completely and unambiguously replaced by representing the relations between the membrane's contents. According to Paun's membrane definition, the size and the spatial placement (distances, positions with respect to any coordinates) do not matter [14, p. 51]. We therefore decided to dispose the membranes in 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.

Relations between the membranes are basically only used when objects are being transferred between two membranes. These relations have been replaced in our implementation by electronic buses (a collection of wires). A membrane can 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 and not with 2 and 3. Therefore, connections in the form of bi-directional communication channels must exist between a membrane, its lower-immediate membranes and upper-immediate membrane (see Figure 2, A). In the following, the two communication directions shall be described.

Firstly, one communication channel, i.e., a bus is required to *send* objects to the lower-immediate membranes (see Figure 2, 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.

Secondly, to *receive* objects from the lower-immediate membranes one bus for each lower-immediate membrane would basically be required since objects can be received simultaneously (Figure 2, C). We did not adopt this solution as it 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., 2 and 3) before it gets connected to the 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 2, D, for example). Therefore, each membrane must basically possess the connections required to connect to both of these levels, although some of them might not always all be used (Figure 2, E and F).

Our architecture is completely independent of the surrounding membrane structure.

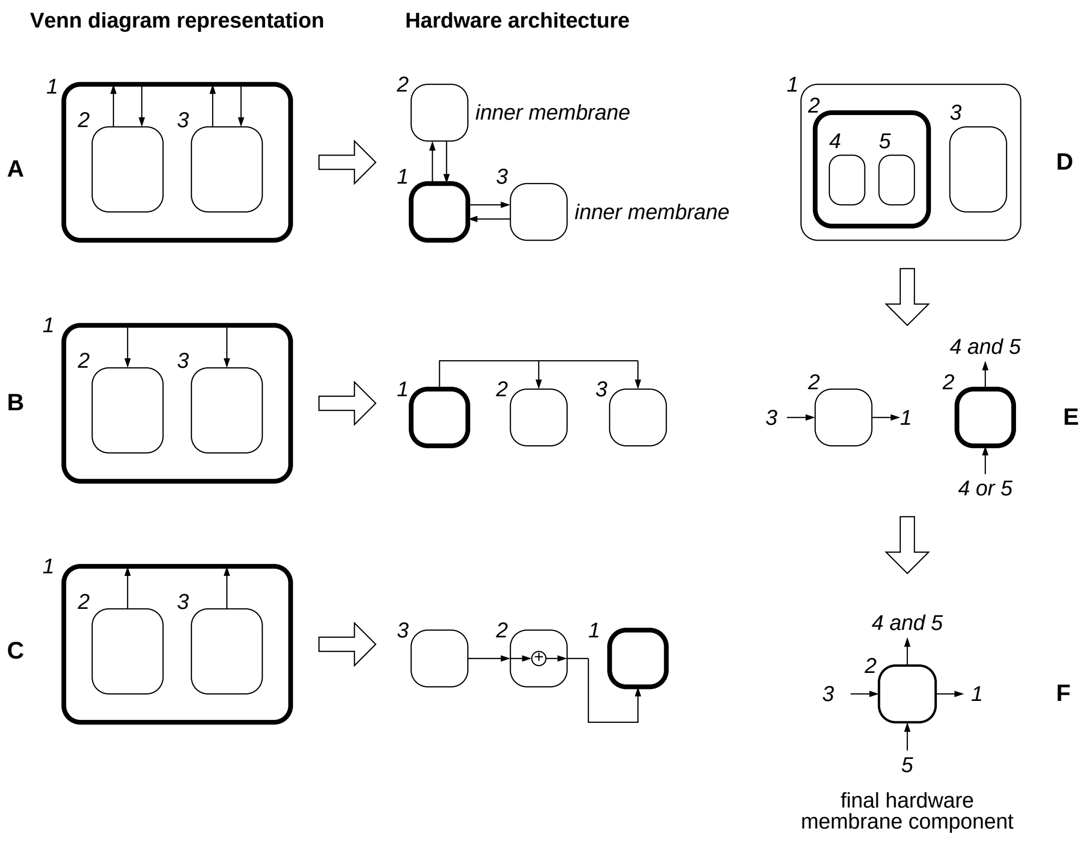


Figure 2: Possible connection structures between membranes. Sub-figure F shows the final membrane hardware component with its interconnection structure. The membrane component is universal and might be used anywhere in a membrane system.

For example, as shown in Figure 2, A, the number of interconnections was 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 2, F – is a sort of universal component that might be used anywhere in the membrane system.

The size of the buses is equal to the size of the multiset of objects which therefore allows to transfer all objects at the same time in parallel (as opposed to a serial communication).

3.2 Multisets of Objects

Each region of a membrane can potentially host an unlimited number of symbol-objects. In our case, these objects are not implemented and represented individually but only their *multiset* is stored. Each object is stored within a register by its multiplicity as shown in Figure 3. The order of the registers reflects the order of the alphabet, i.e., the register position directly indicates which symbol of the alphabet is being stored. It is easy to see that the size of the register limits the maximum number of object instances that can be stored. In Figure 3, for example, it is possible to store up to $2^8 = 256$ multiplicities of an object. In our implementation, the seventh bit is used to detect a capacity overflow, and it is therefore possible to store only $2^7 = 128$ instances of an object. However, the register size can easily be specified at compile time (generic parameter).

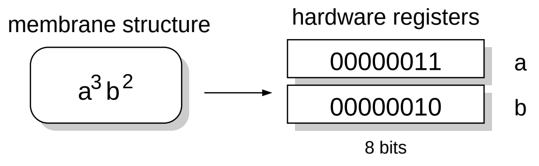


Figure 3: Representation of the symbol-objects in hardware. $\mathsf{Alphabet} = \{\mathsf{a}, \mathsf{b}\}$

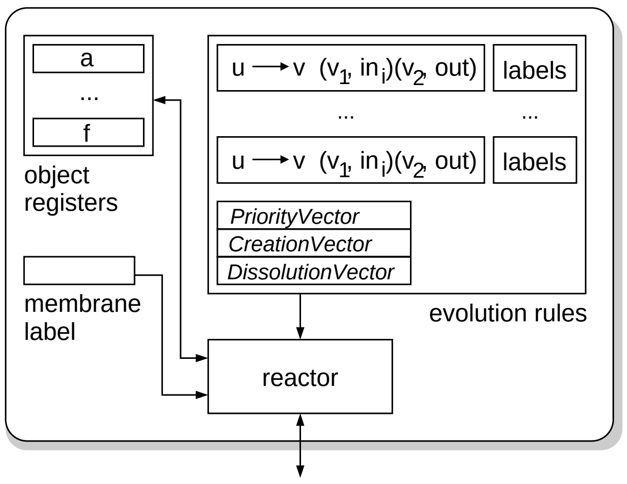


Figure 4: A view within the membrane hardware component.

3.3 Evolution Rules

An evolution rule is a rewriting rule of the form $u \to v(v_1, in_i)(v_2, out)$ that replaces the membrane objects u by the objects v, sends the objects v_1 to the lower-immediate membrane i, and sends the objects v_2 to upper-immediate membrane. The evolution rule is restricted to one target (there are several in general) in our implementation, but this could be rather easily changed. It even appears that rules that are restricted to one target allow for universal computation too.

The module that is responsible for the evolution rules (see also Figure 4) is composed of the following sub-components:

- 4 register arrays (multisets of objects) to store the *left-hand side, here, out* and *in* (i.e., u, v, v_1, and v_2) parts of each rule;
- an eight-bit register that contains the label of the *in* target command (i.e., i from in_i);
- one bit of the *DissolutionVector* that indicates whether the rule will dissolve the membrane;
- one bit of the *PriorityVector* that is set to “1” if there are rules with higher priority;
- four eight-bit registers that store the priority dependency labels;
- one bit of the *CreationVector* that indicates whether the rule will create a membrane; and

- an eight-bit register that specifies the label of the membrane to be created.

All these registers are initialized at the beginning of the simulation that do not change anymore afterwards. Remember also that a P system's evolution rules do basically not change throughout a computation at the exception of a computation with dissolving membranes where the associated set of rules disappears.

In addition,

- a module constantly computes whether a given rule can be applied. The module gets the membrane objects as inputs and generates the signal *Applicable*.

$$\mathit{Applicable}\,(u \to v) = 1 \qquad \text{iff} \qquad u \leq w \tag{1}$$

where $u > 0$ and w is the multiplicity of membrane objects. For example, the rule $a^3 \to b$ can only be applied if there are at least 3 objects of a present within the current membrane.

3.4 Reactor Algorithm

The evolution of a P system can basically be decomposed into *micro-* and *macro-steps*. Algorithm 1 shows the micro-step of our hardware implementation. The "store" operation simply updates the internal registers. A macro-step consists in applying the micro-steps in parallel to all membranes as long as a rule can be applied somewhere. At end of the macro-step (i.e., no rule is applicable anywhere), the received and stored objects will be consolidated and added to the already existing membrane objects in all membranes. The system knows when all micro-steps have been completed due to a global signal that is generated on the basis of the membrane's *Applicable* signals.

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 ($v$)
    Send $v_1$ to the lower-immediate membranes
    Store $v_2$
end if
Receive and store objects from upper-immediate membrane
```

The selection of the rules (first step of Algorithm 1) is done in the following way: a special memory contains the list of rules given in a certain order (specified at the P system initialization). This sequence might either be generated randomly – which simulates in a certain sense a non-deterministic 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 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 $r_1 : u_1 \to v_1$ from a set R_i is used in a transition step with respect to Π (the membrane system) if there is no rule $r_2 : u_2 \to v_2$ which can be applied at the same step and $r_2 > r_1$ [14, p. 70] (i.e., r_2 has a higher priority than r_1).

Remember that there is a 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:

$$PriorityApplicable\,(r_i) = \neg \left(\sum_j Applicable\,(r_j) \mid r_j > r_i \right) \tag{2}$$

The information $r_j > r_i$ is contained in the *labels*. 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 P systems can solve NP-complete problems in linear time (see for example [14, p. 281]).

The first extension is the *membrane dissolving action* [14, p. 64]. The application of rules of the form $(u \to v)\,\delta$ 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.

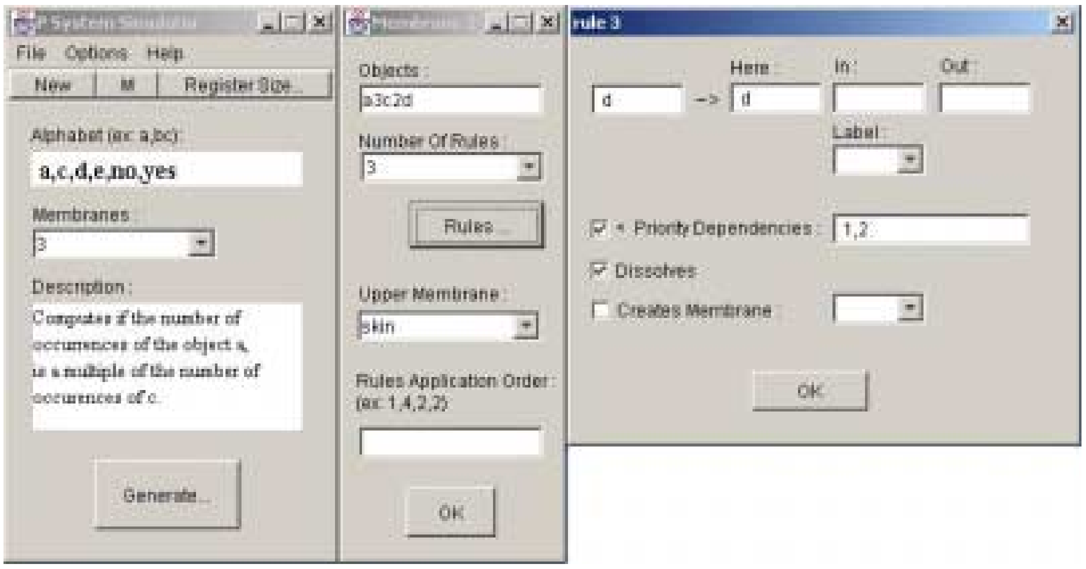
The solution adopted for our hardware implementation simply consists in disabling the membrane by means of a membrane *Enable* signal. Once a membrane dissolving rule has been used, the membrane objects are sent to the upper-immediate membrane, then, the enable signal is set to "0" to disable the membrane. It is important to note that the membrane (i.e., its contents) continues to physically exist. This is, among other reasons, necessary to drive 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 [14, p. 302]. When a rule of the form $a \to [_i v]_i$ 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 apply 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 rules has been used, one of the already created membranes will simply be activated. Obviously, the number of available inactive spare membranes limits the computational power of the system.

4 A Java-Tool for Generating the VHDL Code

The membrane system has been programmed using the VHDL [2] hardware description language. Since a large part of the code is dependent of the membrane system to be simulated and its initial values, a Java application is provided to facilitate the adaptation of the VHDL source code. As shown in Figure 5, the application allows to specify 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. Figure 6 illustrates the typical design flow.



Alphabet (ex: a,b,c): a,c,d,e,no,yes

Membranes: 3

Description: Computes if the number of occurrences of the object a, is a multiple of the number of occurences of c.

Objects: a3c2d

Number Of Rules: 3

Upper Membrane: skin

Rules Application Order: (ex: 1,4,2,2)

d -> Here: d In: Out:

Label:

[✓] < Priority Dependencies: 1,2

[✓] Dissolves

[ ] Creates Membrane

Figure 5: Java application that allows to automatically generate all necessary configuration files of the membrane system hardware implementation.

5 Experiments and Results

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

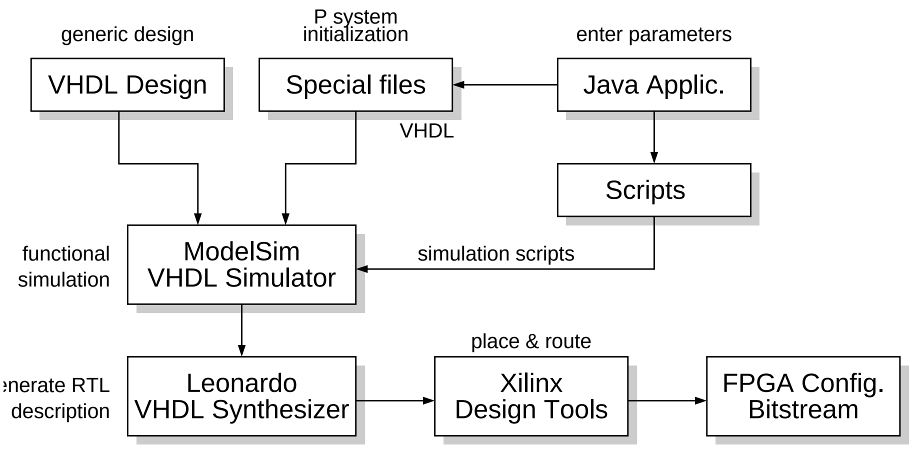


Figure 6: Illustration of the typical design flow.

5.1 An Example

We consider the P system presented on page 73 in [14] that allows to decide whether k divides n. Figure 7 shows the timing diagram of a ModelSim simulation of this P system. The alphabet used is: $\mathsf{Alphabet} = \{\mathsf{a}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{no}, \mathsf{yes}\}$. The signal names to the right (upper-half of the image) represent the following information:

- **clk**: Global clock signal.
- **macro_step**: Indicates when a macro-step is being completed and when the objects in all cells are being consolidated.
- **values i**, and **newvalues i**: Correspond the 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 the rules are not represented in Figure 7 as they do not change. A description of the simulation is as follows (the points basically correspond to the changes in the simulation):

1. The system is initialized. The inner membrane (2) contains a^3b^2d (*values2*, `init`). The system therefore decides whether $n = 3$ divides $k = 2$.
2. Rule $ac \to e$ is applied twice. Two objects of a and c are taken out and replaced by one object d. Note that rule $d \to d\delta$ cannot be applied in this macro-step as a rule of higher priority is applicable.
3. No rules can be applied in any membrane, therefore a macro-step begins and the *values* are updated: *values* := *values* + *newvalues*.
4. The rule $ae \to c$ is applied (the only applicable rule). Again, a macro-step and an update as in the previous step follow.
5. The rule $d \to d\delta$ is used as the rules of higher priority cannot be applied. This causes membrane 2 to dissolve in the next macro-step. Its contents will be transferred to membrane 1.

6. The rule *dce* $\rightarrow$ $(no, in3)$ is applied as it has priority over the other rules. Objects *dce* in membrane 1 are replaced with *no* (`result`) in membrane 3 (the output membrane).

The simulation requires 15 clocks steps to complete.

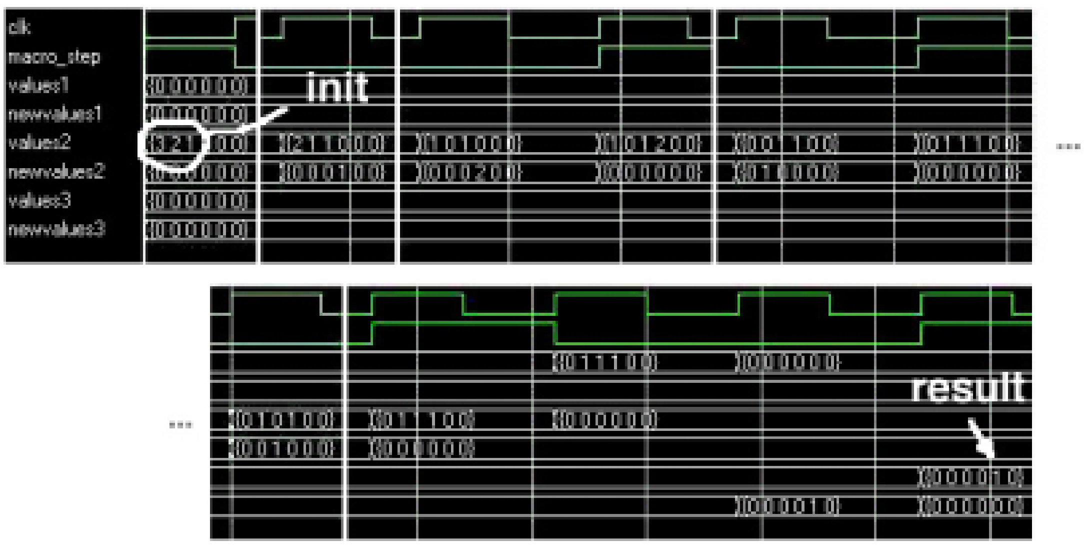


Figure 7: Screenshot of a ModelSim P system simulation. The white fat vertical lines indicate cuts on the time axis, i.e., irrelevant parts of the simulation have been removed.

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 FPGA[3] that roughly contains about 24000 configurable logic blocks (CLBs)[4]. 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.

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.

[3]`http://www.xilinx.com`
[4]1 CLB = 4 logic cells, 1 logic cell = 4-input LUT + FF + Carry Logic.

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 parameterizable 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 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.

The drawback of the proposed implementation is that it is limited to a special class of membrane systems (see also Section 2).

Future work will on one hand concentrate on the development and improvement (in terms of speed and resources used) of the existing design, on the other hand, it would also be necessary to find useful (killer-) applications that require large and complex membrane systems that would fully exploit the specialized hardware. A third approach consists in developing more general hardware architectures for membrane systems amalgamated with alternative computational and organizational metaphors. *Amorphous Membrane Blending (AMB)* [17], for example, is an original and not less unconventional attempt to combine some of the interesting and powerful traits of *Amorphous Computing* [1], *Membrane Computing* [14], *Artificial Chemistries* [4], and *Blending* [5]. The ultimate goal 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] H. Abelson, D. Allen, D. Coore, C. Hanson, E. Rauch, G. J. Sussman, R. Weiss, Amorphous Computing, *Communications of the ACM*, **43**, 5, May 2000, 74–82.

[2] P.J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[3] G. Ciobanu, D. Paraschiv, Membrane Software. A P System Simulator, *Fundamenta Informaticae*, **49**, 1–3 (2002), 61–66.

[4] P. Dittrich, J. Ziegler, W. Banzhaf, Artificial Chemistries–A Review, *Artificial Life*, **7**, 3 (2001), 225-275.

[5] G. Fauconnier, M. Turner, *The Way We Think: Conceptual Blending and the Mind's Hidden Complexities*, Basic Books, 2002.

[6] S. De Franceschi, L. Kouwenhoven, Electronics and the Single Atom, *Nature*, **417**, June 13 2002, 701–702.

[7] L. Garber, D. Sims, In Pursuit of Hardware-Software Codesign, *IEEE Computer*, **31**, 6, June 1998, 12–14.

[8] C.G. Langton, Self-reproduction in cellular automata, *Physica D*, **10**, 1984, 135–144.

[9] M. Madhu, V. S. Murty, 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.

[10] D. Mange, M. Sipper, A. Stauffer, G. Tempesti, Toward Robust Integrated Circuits: The Embryonics Approach, *Proceedings of the IEEE*, **88**, 4, April 2000, 516–540.

[11] N. Mathur, Beyond the Silicon Roadmap, *Nature*, **419**, 6907, October 10 2002, 573–575.

[12] J. von Neumann, *First Draft of a Report on the EDVAC*, Moore School of Electrical Engineering, University of Pennsylvania, June 30 1945.

[13] Gh. Păun, G. Rozenberg, A Guide to Membrane Computing, *Journal of Theoretical Computer Science*, **287**, 1, 2002, 73-100.

[14] Gh. Păun, *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, Heidelberg, 2002.

[15] Gh. Păun, Computing with Membranes, *Journal of Computer and System Sciences*, **61**, 1, 2000, 108–143 and *TUCS Research Report*, **208**, November 1998, `http://www.tucs.fi`.

[16] C. Teuscher, D. Mange, A. Stauffer, G. Tempesti, Bio-Inspired Computing Tissues: Towards Machines that Evolve, Grow, and Learn, *BioSystems*, **68** 2–3, February–March 2003, 235–244.

[17] C. Teuscher, *Amorphous Membrane Blending*, unpublished whitepaper, 2003.

[18] A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, A.E.P. Villa, POEtic Tissue: An Integrated Architecture for Bio-Inspired Hardware, *Evolvable Systems: From Biology to Hardware*, Proceedings of the 5^{th} International Conference (ICES2003) (A.M. Tyrrell, P.C. Haddow, J. Torresen, eds.), Lecture Notes in Computer Science **2606**, Springer-Verlag, Berlin, Heidelberg, 2003.

[19] F. Varela, H. Maturana, R. Uribe, Autopoiesis: The Organization of Living Systems, its Characterization and a Model, *BioSystems*, **5**, 1974, 187–196.

[20] A.H. Veen, Dataflow Machine Architecture, *ACM Computing Surveys*, **37**, 4, December 1986, 365–396.

[21] J. Villasenor, W.H. Mangione-Smith, Configurable computing, *Scientific American*, **276**, 6, June 1997, 54–59.

[22] *International Technology Roadmap for Semiconductors*, Semiconductor Industry Association, `http://public.itrs.net/Files/2001ITRS`, 2001.