A Distributed Simulation of Transition P Systems

Apostolos Syropoulos, Eleftherios G. Mamatas,
Peter C. Allilomes, and Konstantinos T. Sotiriades

Research Division
Araneous Internet Services
366, 28th October Str
GR-671 00 Xanthi, GREECE
research@araneous.gr

Abstract. P systems is a new model of computation, inspired by natural processes, that has a distributive nature. By exploring this distributive nature of P systems, we have built a purely distributive simulation of P systems. The simulation, whose implementation is described here, was programmed in the Java programming language and makes heavy use of its *Remote Method Invocation* protocol. The class of P systems that the simulator can accept is a subset of the $\mathrm{NOP}_2(\mathrm{coo},\mathrm{tar})$ family of systems, which have the computational power of Turing machines. The paper concludes with some remarks concerning the usefulness of the simulation. In addition, there is a brief discussion of some ideas that can be used in the formulation of a foundation of distributive computing.

Keywords: P systems, Natural computation, Distributed Computing, Java's Remote Method Invocation, Object-oriented programming, and Simulation.

1 Introduction

Nature is a constant source of inspiration for artists and scientists. Computer scientists are no exception as they have devised new computational paradigms that imitate natural processes. Such new computational paradigms include DNA computing [8], evolution computing [6], and membrane computing [7]. Nowadays, computational paradigms inspired by natural phenomena and/or processes are collectively known as *natural computation*.

Membrane computing, in particular, was inspired by the way cells live and function. Roughly speaking, a cell consists of a membrane that separates the cell from its environment. In addition, this membrane consists of compartments surrounded by membranes, which, in turn, may contain other compartments, and so on. At any moment, matter flows from one compartment to any neighboring one. Obviously, at any moment a number of processes occur in parallel (e.g., matter moves into a compartment, while energy is consumed in another compartment, etc.).

A P system is a computational device which is an abstract representation of a particular membrane structure. Each compartment is populated by a multiset[1] of symbols. These multisets are materialized as strings of symbols. In addition, each compartment is associated with a set of rewriting rules. These rules are applied to the multisets (strings of symbols) of certain compartments and, consequently, change the system's configuration. The rules are applied simultaneously by observing the so-called *maximal parallelism* principle, that is the rules are selected in such a way that only "optimal" output is yielded. When it is not possible to apply any rule, the P system halts. A designated compartment, called the *output* compartment, contains the output of the computation, which is equal to the cardinality of the multiset contained in it.

P systems have the computational power of Turing machines. In particular, it has been proven that very simple systems with only one compartment have the computational power of a Turing machine. It is also a fact that in every P system the activity of each compartment is independent from the activity at any other compartment. Although this is not relevant for the rest of our discussion, we should note that compartments do not share data—they just exchange data.

A distributed computer system is a computer system in which several interconnected computers share the computing tasks assigned to the system. However, each computer has a designated rôle in the overall computational task. This rough description is reminiscent of P systems and their functionality. Indeed, P systems can be viewed as an abstract model of distributed computing. Naturally, there are many aspects of distributed computing that cannot be described with simple P systems (e.g., secure transmission of data). However, such details are not really important for the formulation of a foundation of distributed computing. Also, it is quite encouraging that other researchers share our view (e.g., see [3]). Here we demonstrate the value of this thesis by describing a distributed simulation of a particular class of P systems. Although the P system bibliography contains many reports on simulations of P systems (for example, see [1,2,4,5]), none of them was implemented as a distributed application. Practically, this means that there is still room for improvements and a number of applications no one has thought before.

Structure of the paper. We start with a semi-formal introduction to the underlying theory of rewriting P systems. Then, we describe tools that can be used to implement distributive algorithms, in general. Next we give a thorough description of our simulation and we conclude with a number of remarks concerning our work and future research directions.

2 Introduction to Theory of P Systems

We start by giving an informal definition of what a transition P system is.

[1] For a thorough description of multisets and their underlying theory see [9].

Definition 1. *A P system is a tuple*

$$\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_m, i_0),$$

where:

(i) *O is an alphabet (i.e., a set of distinct entities) whose elements are called objects.*
(ii) *μ is the membrane structure of the particular P system; membranes are injectivelly labeled with succeeding natural numbers starting with one.*
(iii) *w_i, $1 \leq i \leq m$, are strings that represent multisets over O associated with each region i.*
(iv) *R_i, $1 \leq i \leq m$, are finite sets of rewriting rules (called evolution rules) over O. An evolution rule is of the form $u \to v$, $u \in O^+$ and $v \in O_{\text{tar}}^+$, where $O_{\text{tar}} = O \times \text{TAR}$, $\text{TAR} = \{\text{here}, \text{out}\} \cup \{\text{in}_j | 1 \leq j \leq m\}$.*
(v) *$i_0 \in \{1, 2, \ldots, m\}$ is the label of an elementary membrane (i.e., a membrane that does not contain any other membrane), called the output membrane.*

Note that the keywords "here," "out," and "in$_j$" are called *target* commands. Given a rule $u \to v$, the length of u (denoted $|u|$) is called the radius of the rule. A P system that contains rules of radius greater than one is a system with cooperation.

The rules that belong to some R_i are applied to the objects of the compartment i synchronously, in a non-deterministic maximally parallel way. Given a rule $u \to v$, the effect of applying this rule to a compartment i is to remove the multiset of objects specified by u and to insert the objects specified by v in the regions designated by the target commands associated with the objects from v. In particular,

(i) if $(a, \text{here}) \in v$, the object a will be placed in compartment i (i.e., the compartment where the action takes place)
(ii) if $(a, \text{out}) \in v$, the object a will be placed in the compartment that surrounds i
(iii) if $(a, \text{in}_j) \in v$, the object a will be placed in compartment j, provided that j is immediately inside i, or else the rule is not applicable

The m-tuple of multisets of objects present at any moment in the m compartments of a P system constitute a *configuration* of the system at that moment. The m-tuple $(w_1, \ldots, w_m)$ is the *initial* configuration. Given two subsequent configurations C_1 and C_2, we write $C_1 \Longrightarrow C_2$ to denote a *transition* from C_1 to C_2. A sequence of transitions of a P system Π is called a *computation* with respect to Π. A successful computation is one that halts. The result of a successful computation is the number of objects found in the output membrane after the termination of the computation. The set of numbers computed by a system Π is denoted by $N(\Pi)$.

In general, $\text{NOP}_m(a, \text{tar})$ denotes the family of sets of natural numbers of the form $N(\Pi)$, generated by symbol-object P systems of degree at most $m \geq 1$, using rules of type a. Here we use the same notation for the class of P systems of degree at most $m \geq 1$, using rules of type a. In particular, if $a = \mathrm{coo}$, then we have systems with cooperation. Other possible values are "ncoo" (for non-cooperative), "cat" (for catalytic), etc. Although the systems $\mathrm{NOP}_m(\mathrm{coo}, \mathrm{tar})$ are simple enough, they are very powerful. In fact, they can compute anything a Turing machine can. Since systems $\mathrm{NOP}_1(\mathrm{coo}, \mathrm{tar})$ are far too simple, we opted to provide a simulation for a subset of $\mathrm{NOP}_2(\mathrm{coo}, \mathrm{tar})$ systems (see Table 1 on page 363).

3 Tools for Distributed Programming

There are two basic ways to implement a distributed algorithm: (a) on a purely distributed platform or (b) using some network protocol to connect a number of computers that interchange data. Currently, distributed operating systems are not widely available, in general. In addition, those that are widely available, like Plan 9 from Bell Labs (see `http://www.cs.bell-labs.com/plan9dist/`), have a quite limited hardware compatibility list, thus making essentially impractical the development of distributed applications on such systems. Fortunately, all modern general purpose operating systems provide the necessary network capabilities that can be utilized to create distributed applications.

When implementing a distributed algorithm using the second approach described above, one can either follow the peer-to-peer or the client-server architecture. In a client-server architecture clients interact with a server, while there is no direct interaction between clients. On the other hand, in the peer-to-peer architecture all parts can act as either servers or clients, thus permitting the direct interaction between all participating computers. It is quite pedagogical to think how typical peer-to-peer file-sharing applications (e.g., GoZilla, WinMX, Kazaa, etc.) operate—each computer can connect to any other computer in order to download files, while all other computers are permitted to download files from it.

Sockets are the fundamental tool for the implementation of TCP/IP networking applications. However, when building a system upon the peer-to-peer architecture, it is absolutely necessary to have a (new?) network protocol, which will be used for the exchange of data. Unfortunately, socket programming is error prone, while the design and implementation of a new network protocol is not the easiest thing in the world. This means, that one has to resort to existing well-thought solutions. Implementing a peer-to-peer application using Java's *Remote Method Invocation* (or RMI for short) is an excellent choice. The RMI protocol enables an object running on one Java Virtual Machine (JVM) to invoke methods on an object that is running on another JVM. When such an object is invoked, its arguments are "marshalled"[2] and are sent from the local JVM to the remote one, where the arguments are "unmarshalled." When the method terminates, the results are marshalled from the remote machine and are

[2] The terms *marshalled* and *marshalled* are Sun Microsystems, Inc., lingo and refer to the packing and unpacking of data so they can be safely transported from one JVM to another JVM.

sent back to the caller's JVM. If for some reasons an exception is raised, the exception is indicated to the caller.

We have put under serious consideration all the facts above and so we have opted to implement our simulation using Java's RMI, mainly due to its simplicity and power. Naturally, there are other ways to implement such a system. For example, one can use Java's API for XML-Based RPC (JAX-RPC). In this model, remote procedure calls are represented using an XML based protocol. Such a protocol is the Simple Object Access Protocol, or SOAP for short. SOAP is a lightweight protocol for exchange of structured and typed information between peers in a decentralized, distributed environment. One direct benefit of this approach is that the peers that participate in the simulation can be programmed in any language that supports the protocol and not just Java.

4 The Simulation in Detail

As we have already explained, our simulation has been implemented in the Java programming language. In addition, we would like to stress that we opted to use this language for its extremely rich Application Programming Interface (API) and its object orientation.

Initially, we install a copy of our simulator on a number of (different) computers. Randomly, we choose a computer and assign to it the rôle of the external compartment, while the others play the rôle of the internal compartments. Upon start-up, on each computer a `Membrane` object (see Figure 3 on page 364) is ready to participate to the network. When the systems kicks off, the object that has the rôle of the external compartment, reads the specification of a P system from an external text file and stores the data in a `Data` object (see Figure 2 on page 363). The specification should be written in a notation whose grammar (in Wirth's EBNF) is shown in Table 1. The class `Parser` reads and analyzes the input file and stores the input data (see Figure 1 on page 362).

Note that the number that accompanies the keyword `maximum` denotes the maximum number of cycles the system may go. This artificial parameter was introduced in order to prevent a system from going into an infinite loop. Thus, one should be careful when setting this parameter as it may alter the outcome of the computation (e.g., by forcing a premature termination).

As is evident, one can specify any number of compartments in a P system specification. But this does not necessarily mean that the available resources are enough for the simulation. Thus, when the simulator has successfully parsed the P system's specification, the main object decides whether there are enough resources or not. If the available resources match the requirements set by the description of the P system, the simulator starts the computation. Otherwise, it aborts execution. In order to be able to make this decision, the simulator has been designed in such a way that all objects-compartments send multicast UDP packets to a well-known multicast address. Each packet contains the IP address of each sender. Multicast packets are received by every object participating in the "network." Thus, each computer knows which computers are "alive" at any time.

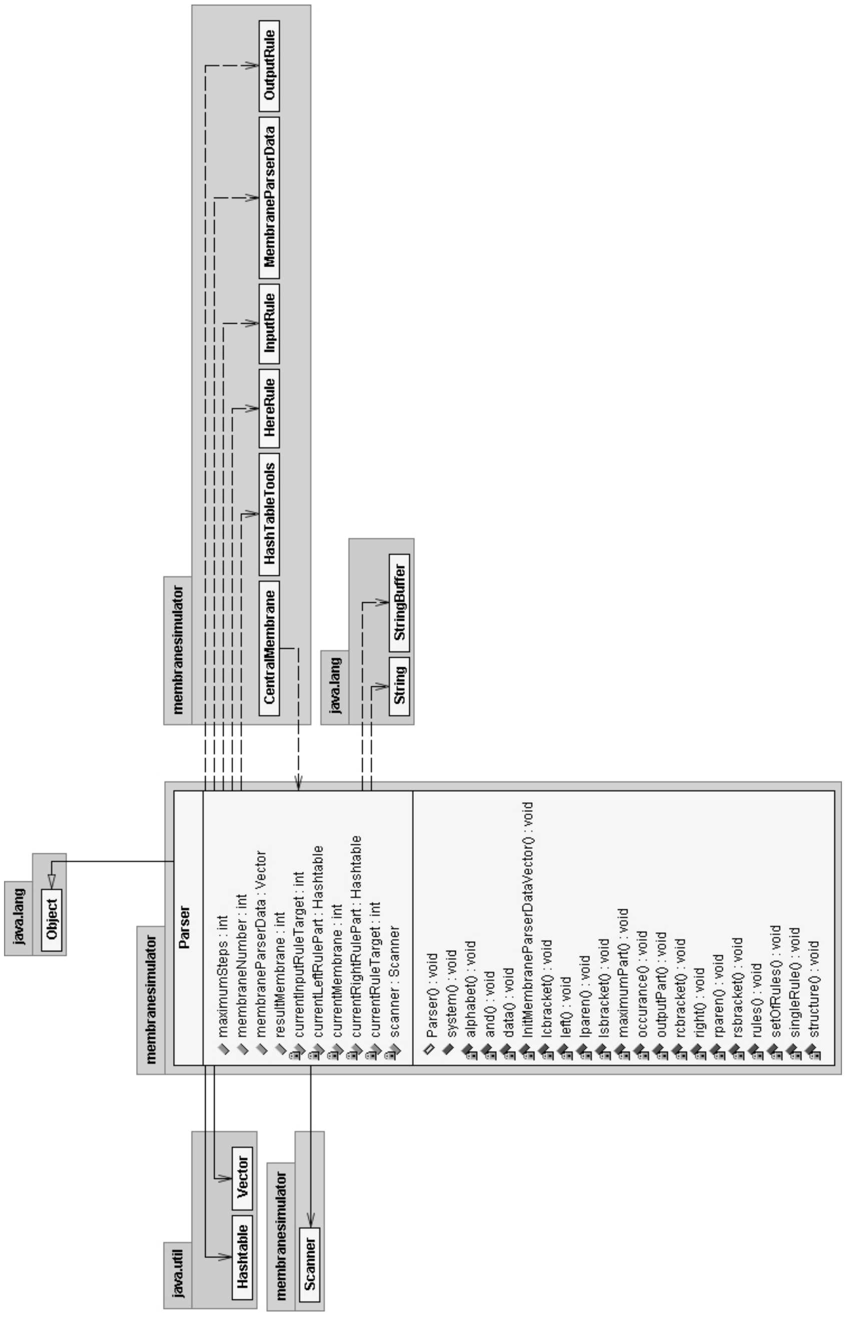

Fig. 1. UML class diagram of the `Parser` class of the simulator.

Table 1. The concrete syntax employed to specify P systems.

system	=	“system” “is”
		alphabet “and”
		structure “and”
		rules “and”
		data “and”
		output “and”
		maximum “and”
		“end”
alphabet	=	“[” letter { “,” letter } “]”
structure	=	“[” { “[” “]” } “]”
rules	=	“{” setOfRules { “,” setOfRules } “}”
setOfRules	=	“[” singleRule { “,” singleRule } “]”
singleRule	=	left “->” right
left	=	letter { letter }
right	=	replacement { replacement }
replacement	=	“(” letter [“,” destination] “)”
destination	=	“here” | “out” | in
in	=	“in” positive-integer
data	=	“{” Mset { “,” Mset } “}”
Mset	=	“(” { occurrence } “)”
occurrence	=	“[” letter “,” positive-integer “]”
output	=	“output” positive-integer
maximum	=	“maximum” positive-integer

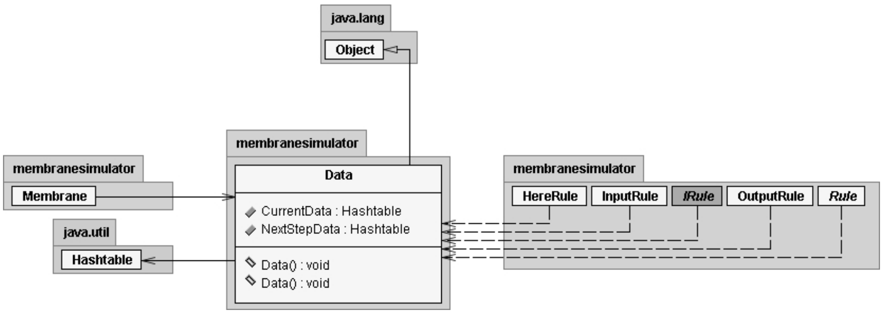

Fig. 2. UML class diagram of the `Data` class that is used to store the (initial) data of a `Membrane` object.

This way the main object has all the necessary information to decide whether there are sufficient resources to start the computation. A universal clock is owned by the object that has the rôle of the external compartment. This object signals each clock tick by the time the previous macrostep is completed (i.e., when, for

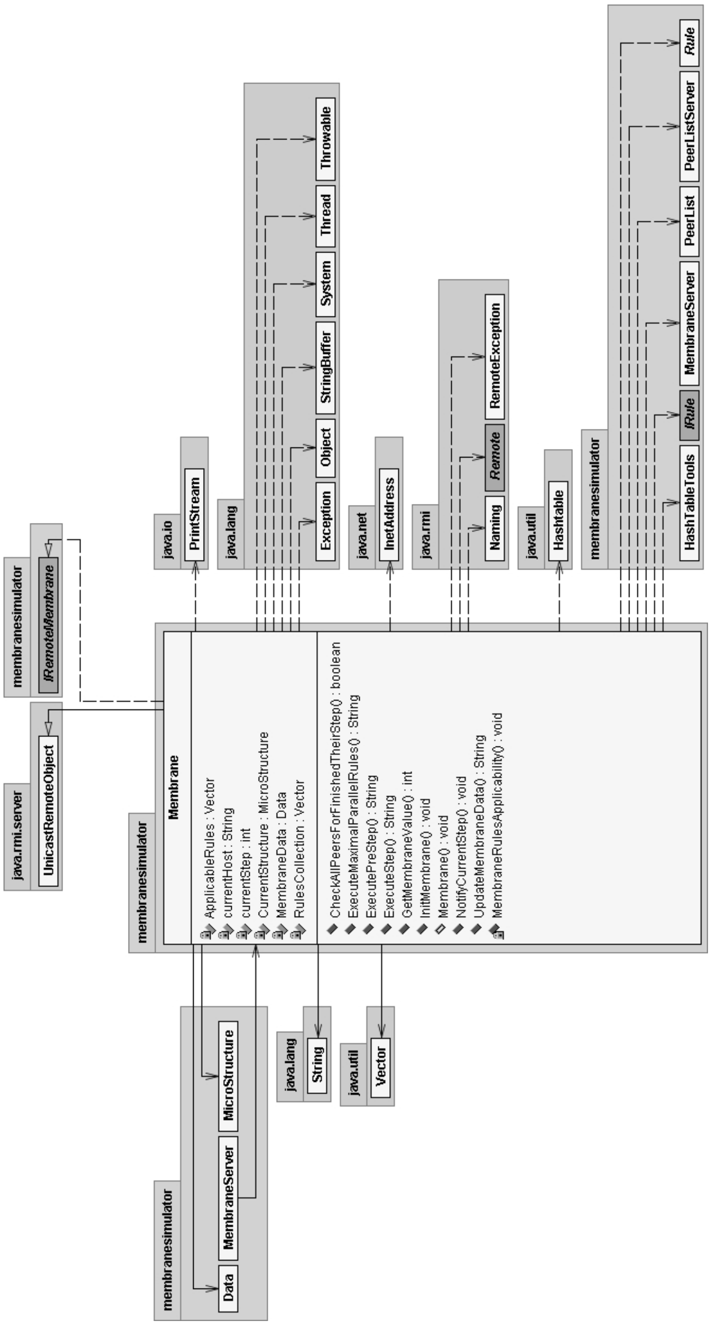

Fig. 3. UML class diagram of the `Membrane` class.

a given macrostep, all remote objects have finished their computation). While the computation proceeds, if there is a communication breakdown with any computer participating in the network, there is a fault tolerance mechanism that takes care of this problem. The action that may be taken varies according to the given situation. For example, if the "missing" compartment does not affect the global computation, then the system proceeds without any problem. The system halts once there are no more applicable rules or the maximum number of cycles has been reached. The following set of rules are applied at each macrostep to gain maximal parallelism.

(i) Initially, the simulator checks which rules are applicable and selects them.
(ii) If there are applicable rules with common elements on their left-hand side, we compute their "weights" as follows:
   (a) We mark the common elements on both sides of each rule
   (b) We remove these common elements from both sides of each rule
   (c) The "weight" of each side of a rule is equal to the number of elements. In case there are no elements left, the weight is equal to one.
   (d) The total "weight" of a rule is equal to the product of the two "weights" for each rule.

   Finally, we select only the rule that has the highest "weight."
(iii) The rules that remain after this selection procedure, are used in the actual computation.

We should note that a rule is applicable if its left-hand side contains elements that are inside the compartment the rule is supposed to be applied. In Figure 4 the reader may inspect the UML class diagram of the `Rule` class, which implements the functionality just described.

Let us now describe in detail what is going on in each macro step:

(i) The object that has the rôle of the outer membrane sends a message to all other objects. This message indicates to them that they have to start the execution of their microstep computation. While the main object starts its own computation, it waits each object to send a message indicating the termination of its micro-step computation.
(ii) The microstep computation involves the application of the rules associated with each object (compartment). In case, some rule demands the transfer of data to some other (remote) object, these data are not send immediately. Instead, they are kept in some buffer that is part of the objects data area.
(iii) When all objects have finished, the main object sends a request to all participating objects to send the data the have accumulated in their corresponding buffers.
(iv) Once all information have reached each corresponding recipient, the system is ready to repeat this cycle and go on to next macrostep.

Threads, that is parts of a program that can execute independently of other parts of the program, are an essential aspect of our implementation. In particular, each membrane class runs in its own thread, which, in turn, operates on a different

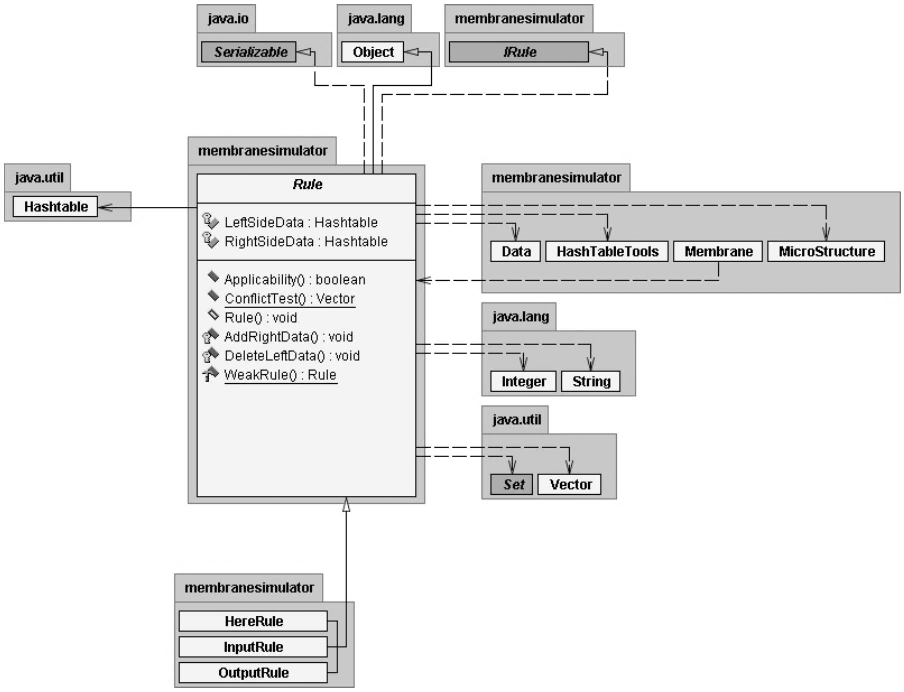

Fig. 4. UML class diagram of the `Rule` class that is used to process the rules.

machine. A fortunate consequence of this implementation is that the system follows very closely the theoretical model of P systems.

The source code of the system, a jar file, which can be used to immediately install the simulator, as well as the documentation of the simulator can be downloaded from our Web site at `http://research.araneous.com`.

5 Conclusions and Future Research

We have presented a purely distributed simulation of a class of rewriting P systems. The simulator was implemented in the Java programming language and was build upon the Remote Method Invocation protocol. Currently, the simulator can be used as an instructive tool that shows all the activity that takes place in a typical P system. In addition, it can be used to study very simple organisms, which can be simulated by P systems that are models of such organisms.

Naturally, our simulator is far from being complete. It is obvious that there is plenty of room for improvements and extensions. For example, it would be of great interest to extend the simulator so it can handle P systems with a larger number of membranes, which may not necessarily use rewriting-like rules to process multisets. Such an extension would appeal biologists, in particular, and people working on mathematical models of living beings, in general. Thus, researchers would have at their disposal a test bed for the evaluation of mathematical models of living organisms.

In spite of these great potentials, we do not consider this aspect of our work as its main outcome. Our simulator has shown to us in a very clear way that P systems are distributive in nature and, thus, they can be used in the formulation of a foundation of distributive computing. Certainly, one may object to this idea by remarking that P systems cannot capture all aspects of distributive computing. However, we believe that this is not a serious drawback, as, for example, Turing machines, which are an abstract model of sequential computing, lack a number of features that are present in all modern computers. In order to test the suitability of P systems as a foundation of distributive systems, one has to study the degree to which P systems can describe the functionality of a distributive architecture. A particularly interesting computer architecture that can be used for this purpose is the *Distributed Instruction Set Computer* (DISC) architecture [10]. The authors of this paper had some preliminary discussions with the designers of the DISC architecture about these ideas. It is quite encouraging to report that there is a consensus among the two teams that this is indeed a promising research direction. An unexpected outcome of such an endeavor would be the fact that one would possibly design a basic distributive instruction set computer similar to the Random Access Machines of classical computing. Obviously, such a development would pave the road for the design of compilers that would be able to compile programs written in some "ordinary" programming language directly to a distributive architecture. This way, one would be able to create applications for distributive architectures without any need to get trained in distributive computing. Note that this is not a novel idea in the field of programming language implementation. For example, the implicit parallelism inherent in functional programs has driven many researchers to implement functional programming languages on parallel architectures [11]. Thus, functional programmers can create programs that are executed on parallel architectures as if they had a parallel design. This means that programmers create "parallel" programs without any need to actually do any parallel programming.

We believe that the theory of P systems is mature enough and that there is no need for any further generalizations. Instead, we need to focus on "real world" application of the theory. We hope that our work and our ideas is a step towards this direction.

References

1. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. Mingo, Structures and Bio-language to Simulate Transition P Systems on Digital Computers, *Multiset Processing* (C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), Lecture Notes in Computer Science **2235**, Springer-Verlag, Berlin, 2001, 1–15.

2. F. Arroyo, C. Luengo, A.V. Baranda, L.F. de Mingo, A Software Simulation of Transition P Systems in Haskell, *Membrane Computing* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), Lecture Notes in Computer Science **2597**, Springer-Verlag, Berlin, 2003, 19–32.
3. G. Ciobanu, R. Desai, A. Kumar, Membrane Systems and Distributed Computing, *Membrane Computing* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), Lecture Notes in Computer Science **2597**, Springer-Verlag, Berlin, 2003, 187–202.
4. G. Ciobanu, D. Paraschiv, A P System Simulator, *Technical Report* **17/01**, Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, Spain, 2001.
5. M. Malita, Membrane Computing in Prolog, *Pre-proceedings of the Workshop on Multiset Processing*, Curtea de Argeş, Romania, **CDMTCS TR 140**, Univ. of Auckland, 2000, 159–175.
6. Zb. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin, 2nd ed., 1994.
7. Gh. Păun, *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, 2002.
8. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
9. A. Syropoulos, Mathematics of Multisets, *Multiset Processing* (C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), Lecture Notes in Computer Science **2235**, Springer-Verlag, Berlin, 2001, 347–358.
10. L. Wang, C. Wu, Distributed Instruction Set Computer Srchitecture, *IEEE Transactions on Computers*, **40**, 8 (1991), 915–934.
11. R. Wilhelm, M. Alt, F. Martin, M. Raber, Parallel Implementation of Functional Languages, in *5th LOMAPS Workshop, Analysis and Verification of Multiple-Agent Languages* June 1997 (M. Dam, ed.), Lecture Notes in Computer Science **1192**, Springer-Verlag, 1997.