Journal of Universal Computer Science, vol. 10, no. 5 (2004), 620-629
submitted: 1/4/04, accepted: 10/4/04, appeared: 28/5/04 © J.UCS

A Java Simulator for Membrane Computing

Isabel A. Nepomuceno-Chamorro
(University of Sevilla, Spain
`Isabel.Nepomuceno@cs.us.es`)

Abstract: Membrane Computing is a recent area of Natural Computing, a topic where much work has been done but still much remains to be done. There are some applications which have been developed in imperative languages, like C++, or in declaratives languages, as Prolog, working in the framework of P systems. In this paper, a software tool (called *SimCM*, from Spanish *Simulador de Computación con Membranas*) for handling P systems is presented. The program can simulate basic transition P Systems where dissolution of membranes and priority rules are allowed. The software application is carried out in an imperative and object-oriented language – Java. We choose Java because it is a scalable and distributed language. Working with Java is the first step to cross the border between simulations and a distributed implementation able to capture the parallelism existing in the membrane computing area. This tool is a friendly application which allows us to follow the evolution of a P system easily and in a visual way. The program can be used to move the P system theory closer to the biologist and all the people who wants to learn and understand how this model works.

Key Words: P system, Parallelism, Simulation, Java

Category: F.1.1, D.0

1 Introduction

Natural Computing contains three well established areas: Neural Networks, Genetic Algorithms, and Molecular Computing. Membrane Computing is a recent area introduced by Gh. Păun where computations are considered at the cellular level.

This new computability model is based on the notion of membrane structure, which consists of several cell-like membranes, recurrently placed inside an external skin membrane. A membrane structure can be representated by a tree or by a Venn diagram without intersected sets. In the compartments defined by membranes there are objects, that can evolve, can be transformed in other objects and can pass through membranes. In each compartment every object can appear a specified number of times, that is to say, the objects form multisets. In the regions delimited by the membranes there are also placed evolution rules that allow to modify the objects and establish communication between membranes. The evolution rules can also dissolve the membranes where they are applied. When a membrane is dissolved, its objects will be left free in the immediately upper membrane (the skin cannot dissolve). In addition, a priority relation between evolution rules can be considered, that is to say, if two rules can

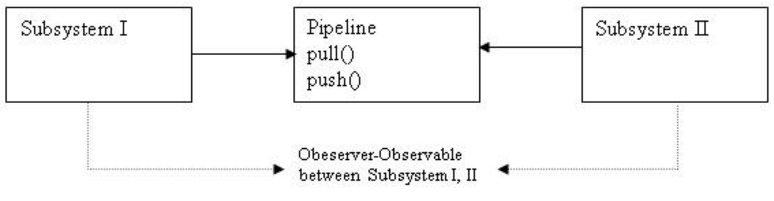


Figure 1: Subsystems of the software.

be applied simultaneously and there is a priority relation between them, then only the higher priority rule can be applied. The evolution rules are applied in a maximal mode, hence in each step of a computation all objects that can be transformed by the rules must be transformed.

The computing device informally described here is called basic transition P system (more details can be found in [Păun 2002] and a formal presentation can be found in [Pérez-Jiménez and Sancho-Caparrini 2002]). The state of a P system in a given step is called a *configuration*; when we apply the evolution rules to a configuration we obtain a *transition*; a *computation* of a P system is a sequence (finite or or not) of transitions between configurations; a computation is *halting* if there is no evolution rule that can be applied in the final configuration.

These devices possess two levels of parallelism: into each membrane, because all the applicable rules should be used simultaneously, and globally, because all the membranes evolve at the same time.

2 A Look Inside the Application

This application we have developed for simulating a P system is a variation of the Model-View-Controller (MVC), an architecture model of software development used in interactive systems where the user interfaces are changeable. It is composed of several different components: in the first one, *Model*, functional qualities and type abstract data are found; the second component, *View*, is responsible for showing the results to the user through a graphical interface; the third component, *Controller*, is in charge of the requests made by the user. In the creation of the SimCM program, the View and Controller components are joined into a single component. Figure 1 picture represents the architecture model of software. Figures 2 and 3 represent the hierarchical structure and the relations between the different Java classes designed and used for the application.

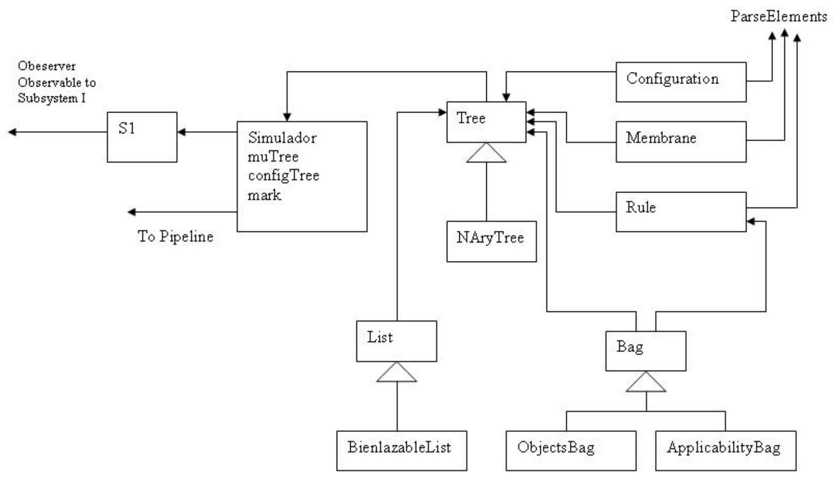


Figure 2: Subsystem I.

These classes are distributed in two subsystems:

– Subsystem I includes the simulator engine, that is, all the functional qualities of the basic transition P systems, and type abstract data. This subsystem is formed by the following Java packages: *NAryTree* (implementing the tree types to represent the membrane structure and the computation tree), *List*, *Membrane*, *Multiset*, *Rules*, and *Simulator*.

– Subsystem II includes all the classes related to the Guide User Interface (GUI) that interacts with the user and is formed by the Java packages: *Interface*, implementing the GUI; *DataUSer*, containing the classes that are in charge of the user's requests for building an initial P system; *ParseRule* contains the classes that are in charge of the lexical analysis of the initial P system introduced by the user; *Serialization*, whose classes allow us to take a step back in the building of an initial P system; and *HelpHtml* that includes the html document.

The two subsystems interact with each other and produce a computation that has been implemented with instance objects of the *Observer* and *Observable* Java classes. The instance objects of the *Observer* Java class should be notified when the state of an *observable* object is modified. The *Observable* objects are any objects whose state can be of interest for any other object (observer). However, the communication between the execution threads of the program can be more difficult than that between the observer and observable. Threads used for com-

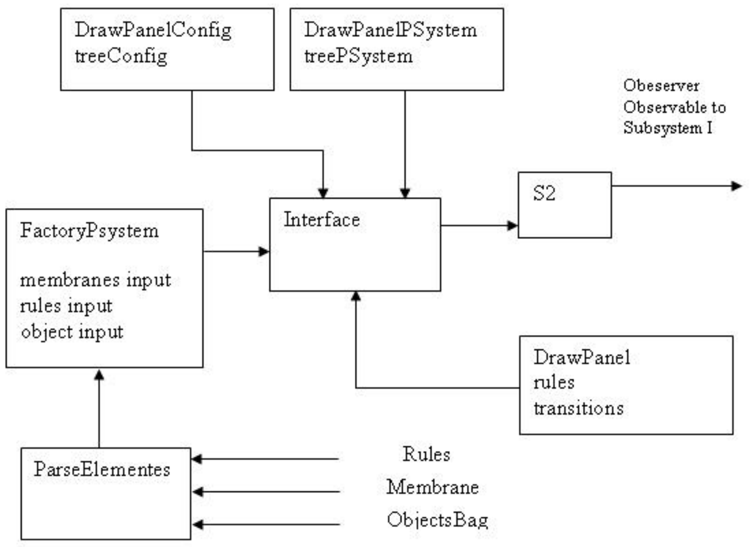


Figure 3: Subsystem II.

mon objects, so that a thread can give an object to another, can be designed in such a way that the objects can be manipulated independently by both sides at the same time. This is the classic example of the thread communication in the producer/consumer problem: one thread, called producer, produces a result that another thread, the consumer, uses or consumes, no matter what the result is. For example, in the program the user can select the *Guided* mode of execution. In this mode, the user selects with the mouse by clicking on the configuration node in the GUI; this node is a product the GUI offers to the simulator in order that the simulator continues running from this node; the simulator engine consumes this object. In order to control this communication, the Java class *Pipeline* has been created, including *push()* and *pull()* functions (synchronized access methods), in order to maintain the integrity of the shared objects.

In the following subsections we describe some programming details of these two subsystems.

2.1 Engine of Simulator

The engine is built upon two fundamental pillars: the first one is the simulator that includes the algorithms to simulate the processes and computations produced inside a membrane system; it also contains the functional qualities of the system, with the task of starting the initial configuration of the P system and constructing the initial configuration of the associated computation tree; the second one includes all the type abstract data in order to support the membrane structure and its content (multisets of objects and rules), and contains the type data necessary for the creation and storage of the applicability multisets.

The two relevant classes of the engine are *Simulator.java* and *CreateBagAplicability.java*.

The class *CreateBagAplicability.java* has the task to create the applicability multiset of rules, *MAp(Ci)*, that must be computed for each configuration. Briefly, the algorithm to obtain *MAp(Ci)* works as follows: for each membrane, the rules are iterated in such a way that in each iteration a distinct rule is pivotal and for each rule its maximum applicability number is calculated according to the multiset associated with the membrane and the other rules are analyzed according to those that can be maximally applied or not (in this way the parallelism of the application rules is simulated, and the distinct possibilities of applying rules for each membrane is obtained); finally, this process continues over each of these possibilities and one obtains the applicable multiset of rules (see [Nepomuceno-Chamorro 2003] for further details).

The main class is *Simulator.java* which extends the *Thread* Java class. In this way an instance of this class can create a context of the system task and execute it through a call to *start* method, stop the execution of the associated thread of the task without destroying it, and execute the run method that starts the simulation algorithm. This class constructs the computation tree associated with the loaded P system: it starts with the initial configuration, computes its applicable multisets of rules and obtains its next configurations; this process is repeated for each configuration until it reaches the final state or a depth level previously established by the user. It must be understood that the efficiency of Simulator class is based on a sound implementation of the data structures, and on the iteration algorithm of these structures. In the construction of the computation tree, the membrane structure associated with each configuration is stored on the hard disk in order to not burden the RAM and not to overload the computer. Finally, *Simulator.java* is controlled by the user in such a way that he/she can re-do the process of the computation tree construction in *one step* or *step by step*. This *step by step* possibility implies a more exhaustive communication with the guide user interface than simply generating the events. In such case we use the *Pipeline.java* class that we have mentioned in the previous section. (See [Nepomuceno-Chamorro 03c] for further details about Simulator engine.)

2.2 Guide User Interface

The GUI is the part of the program allowing the user to interact with the application. The GUI is included in the Subsystem II with the management-reception of events generated between the user and the Simulator engine. The *Swing* Java package is used in the construction of this GUI (an independent platform library of classes used to develop GUI's). This package includes the *AWT*[1] package (Abstract Windows Toolkit) and extends its capabilities. The AWT package was the first library class offered in Java.

Briefly, among the classes that form the GUI, the most important are:

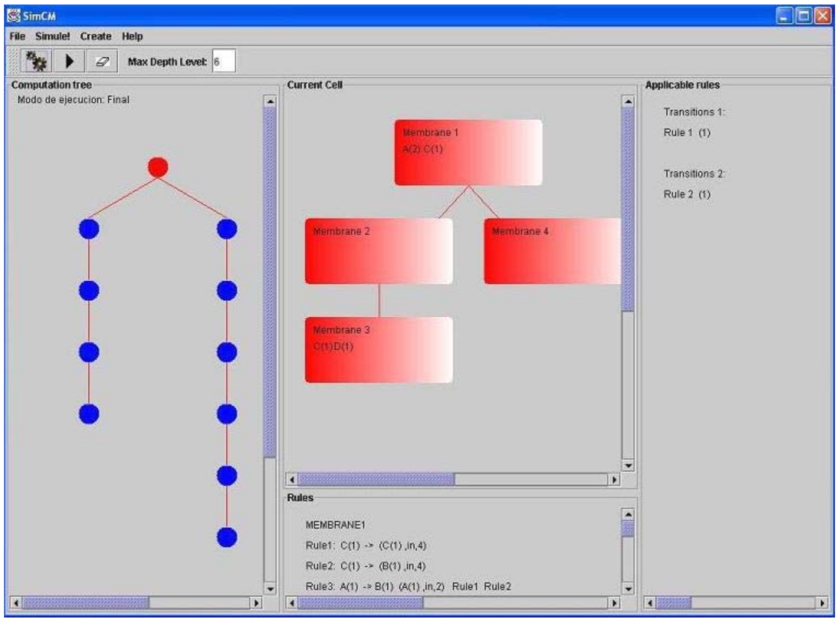
– The classes that are in charge of drawing trees: *GraficaArbolConfig.java* and *GraficaArbolMu.java*, with the task to allow a visual representation of the computation tree and the membrane structure (both of them extend the *JPanel* Java class). In the first case, when the Simulator engine is finished or need to show an intermediate result (in the guided user mode where one can see how the computation tree is builded), this class notifies it to *GraficaArbolConfig.java* and draws it. In the second case, when the user loads or builds a P system or selects a configuration of the computation tree the class *GraficaArbolMu.java* draws the membrane structure in the appropriate panel.

– *Frame.java*; this class extends the Java class JFrame and the former is the container of the menu bar, toolbar and the panel where the drawing of the computation tree and the membrane structure is produced.

– *Building.java* and *Parser.java*. The first class is in charge of building an initial P system with the data inserted by the user. It parses the input to avoid mistakes (for example, when the user introduces new membranes, he/she must indicate the father tag and if this tag does not exist, then the program will advise the user that the action is not allowed). The class *Parser.java* parses the lexical alphabet and the multiplicity of objects of multisets. Finally, when the user introduces new rules, if the rules are associated with non-existing membrane tags or if there are lexical errors in the multisets or non-existing greater priority rule tags, this class warns the user about this.

– *Memento.class* captures and externalizes an object's internal state so that the object can be restored to this state later. In SimCM software the object is the membrane structure.

[1] The AWT is part of the Java Foundation Classes (JFC), the standard API for providing graphical user interfaces (GUIs) for Java programs.



Modo de ejecucion: Final

Transitions 1:
Rule 1 (1)

Transitions 2:
Rule 2 (1)

```
MEMBRANE1
Rule1: C(1) -> (C(1) ,in,4)
Rule2: C(1) -> (B(1) ,in,4)
Rule3: A(1) -> B(1) (A(1) ,in,2)  Rule1  Rule2
```

Figure 4: The main screen of the program.

3 Overview of the Application

Essentially, this software tool allows us to handle transition P systems by means of three basic operations:

– Create an initial membrane system; the simulator includes a *debug mode* in order to avoid some user errors.

– Load and Save previously defined membrane systems.

– Carry out a simulation of the P system evolution. This simulation can be performed in three different ways: until a given maximal level, level by level, and guided.

The guide user interface (GUI) of the application (Figure 4) uses classes of the Java Swing[2] package and the Integrated Development Environment (IDE) used is Forte[3].

The main screen is divided into four basic panels:

[2] Swing is part of the Java Foundation Classes (JFC). This is a set of Java class libraries provided as part of Java 2 Platform, Standard Edition (J2SE), to support building graphics user interface (GUI).

[3] Forte for Java v.3.0 Copyright Sun Microsystems

– Computation tree: this panel shows the tree of configurations after the simulation is finished or during its development.

– Current cell: initially, this panel contains a sketch in the form of a tree representing the membrane structure of the system to be studied (the program represents the membrane structures as tree structures). Once the simulation is finished or when it is in development, this panel will represent the state of the membrane system according to the configuration chosen by the user in the computation tree panel. In order to select the configuration, a simple click on the chosen node in the tree of the computation panel is needed.

– Rules: in this panel the rules associated with each membrane are shown.

– Applicable rules: this panel shows the applicability multiset associated with the configuration selected by the user in the computation tree.

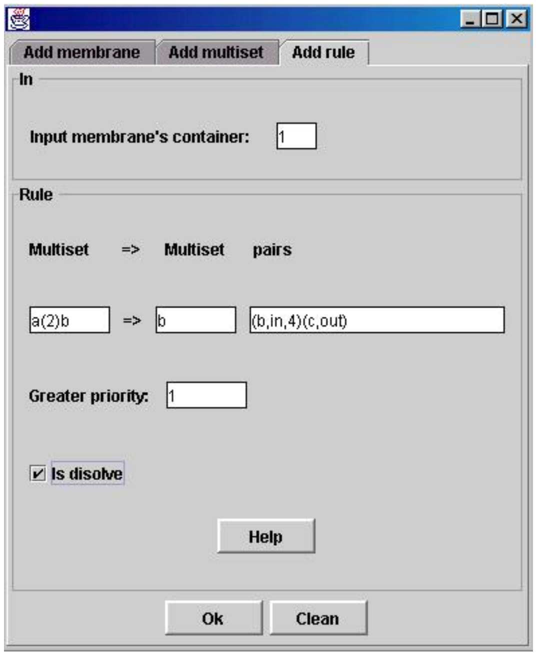
In the menu bar several actions are found; among them, we mention:

1. Create P systems. To do this, an initial membrane system must be created by selecting the insertion of new structures. In this case one must click *Create* and next *New* in the menu and the skin appear in the current cell panel. New elements such as other membranes, rules and objects can be added. There is a form that has been created to help the user in the insertion (Figure 5). The program parses the input as described in Section 2.2.

2. Start the evolution of the P System. There are three modes of simulation: the first one is *Until Max Depth Level*, in which the simulation engine runs and does not show the final result until it either reaches the stop configuration or it hits the max level given by the user. The second mode is *Level by Level*, in which the simulation engine runs and shows the results as they are computed step by step in the computation tree. The third mode is *Guided*, in which with each new level the user has to select the corresponding node according to the configuration the user wants to continue computing, and then click *Next step* in the menu or toolbar.

3. Erase computation tree – in order to clean panels and memory, to work with a new P system.

In the toolbar several actions are found. The most important is the fact that the user can use the text box to set the boundary mark, in order to set the desired depth of the tree of configurations. By default, the mark is set at four levels. In this way, in the case that the stop configuration does not exist, the program will finalize the simulation of the P system when the tree of configurations reaches the mark indicated in the text box.



Input membrane's container: 1

a(2)b => b (b,in,4)(c,out)

Greater priority: 1

[✓] Is disolve

Figure 5: Insertion form.

4 Future Work

We plan to continue our work by developing the tools for handling other types of P systems. The next step could be to implement a distributed P system in Java RMI or using the standard CORBA[4]. In this way we could use the simulator of distributed computing tools to capture the idea of maximal parallelism present in this model, so the future software would be more similar to the way the P systems compute.

Acknowledgements

Thanks are due to Fernando Sancho-Caparrini for the help in completing this work.

The software SimCm, can be downloaded from the page of the Research Group on Natural Computing from Sevilla University (`http://www.gcn.us.es/`).

[4] Common Object Request Broker Architecture

References

[Ciobanu and Paraschiv 2002] Ciobanu G., Paraschiv D.: “Membrane Software. A P System Simulator”; Pre-Proceedings of Workshop on Membrane Computing, Curtea de Arges, Romania, August 2001, Technical Report 17/01 of Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, Spain, 2001, 45–50, and Fundamenta Informaticae, 49, 1-3 (2002), 61–66

[Cordón-Franco et al. 2003] Cordón-Franco A., Gutiérrez-Naranjo M.A., Pérez-Jiménez M.J., Sancho-Caparrini F.:“A Prolog Simulator for Deterministic P Systems with Active Membranes”; Rovira i Virgili Univ., Tech. Rep. No. 26/2003, M. Cavaliere, C. Martín-Vide, Gh. Păun (Eds.), “Brainstorming Week on Membrane Computing”, Tarragona, February 5-11 (2003), 141–154

[Gamma et al. 1995] Gamma E., Helm R., Johnson R., Vlissides J.: “Design Patterns. Elements of Reusable Object-Oriented Software”; Addison-Wesley (1995)

[Grand 1998] Grand M.: “Patterns in JAVA. Vol.1 A Catalog of Reusable Design Patterns Illustrated with UML”; Wiley (1998)

[Hernández at al. 2000] Hernández R., Lázaro J.C., Dormido R., Ros S.:“Estructura de datos y algoritmos“; Prentice Hall (2000)

[Nepomuceno-Chamorro 2003] Nepomuceno-Chamorro I.A.: “Simulaciones de P Systemas en Java, aplicación SimCM”; CCIA Universidad de Sevilla Sección III, N 3, 2003, 1–115

[Păun 2002] Păun Gh.: “Membrane Computing. An Introduction”; Springer, Berlin (2002)

[Pérez-Jiménez and Sancho-Caparrini 2002] Pérez-Jiménez M., Sancho-Caparrini F.: “Computación celular con membranas: Un modelo no convencional”; Kronos, Sevilla (2002)