WebPS: A Web-based P System Simulator with Query Facilities*

Cosmin Bonchiş[1], Cornel Izbaşa[2], Dana Petcu[1,2], Gabriel Ciobanu[1,3]

[1] Research Institute "e-Austria" Timişoara, Romania
Bd. V. Pârvan 4, 300223 Timişoara, Romania
E-mail: cosbonchis@yahoo.com

[2] Department of Computer Science, Western University of Timişoara
Bd. V. Pârvan 4, 300223 Timişoara, Romania
E-mail: {petcu, cornel}@info.uvt.ro

[3] Romanian Academy, Institute of Computer Science, Iaşi
Bd. Copou 8, 700505 Iaşi, Romania
E-mail: gabriel@iit.tuiasi.ro

Summary. In this paper we present an open-source web-enabled simulator for P systems. We use CLIPS embedded in C, and make the simulator available as a web application, complemented by a query language to specify the results.

1 Introduction

Transition P systems and deterministic P systems with active membranes (see [4]) are simulated in various programming languages, and some of them are used to solve NP-complete problems as SAT, Subset Sum, Knapsack, and partition problems. P systems with active membranes, input membrane and external output are simulated in CLIPS, and used to solve NP-complete problems (see [5]). New variants of these simulators provide symport-antiport rules, and catalysts. A more complex simulator written in Visual C++ for P systems with active membranes and catalytic P systems is presented in [2]. It provides a graphical simulator, interactive definition, visualization of a defined membrane system, a scalable graphical representation of the computation, and step-by-step observations of the membrane system behavior.

A parallel and cluster implementation for transition P systems in C++ and MPI is presented in [1]. The rules are implemented as threads. At the initialization phase, one thread is created for each rule. Since each rule is modelled as a separate thread, it should have the ability to decide its own applicability in a particular

* This work has been supported by the Institute e-Austria Timişoara, BMWA and BMB:WK ministries of Austria.

round. In order to detect if the P system halts, each membrane must inform the other membranes about its inactivity. It can do so by sending messages to others, and by using a termination detection algorithm.

In this paper we present a new simulator based on CLIPS. The previous CLIPS implementations represent P systems rules by CLIPS facts; we get a significantly faster execution by using CLIPS rules to implement P systems rules. In [5], the P system data are inserted directly into code, and so they cannot be easily modified, as opposed to our implementation where a P system is an input data, providing a lot of flexibility. P systems as input data are described by XML documents; this fact provides a standard method to access information, making it easier to use, modify, transmit and display. XML is readable and understandable, it expresses metadata easily in a portable format, just because many applications can process XML on many existent platforms. Moreover, by using XML, it becomes easy to define new features and properties of P systems. XML allows an automated document validation, restricting wrong input data, and warning the user before execution with respect to the possible errors in their P systems description. From a user point of view, all these features facilitate an efficient description and reconfiguration of the P systems.

An important advantage of our simulator is given by its availability as an Web application; this aspect is actually emphasized by its name: WebPS. Web applications do not require an installation. As any Web application, it can be used from any machine anywhere in the world, without any previous preparation. A simple and easy to use interface allows the user to supply an XML input both as text and a file. A friendly way of describing P systems is given by a helpful interactive JavaScript-based P system designer. The interface provides a high degree of (re)usability during the development and simulation of the P systems. The initial screen offers an example, and the user may find useful documentation about the XML schema, the rules, and the query language. The query language helps the user to select the output of the simulation.

The simulator is *open-source*, actually *free software*, as it is being offered at `http://psystems.ieat.ro` under the *GNU General Public License*. This allows anyone to contribute with enhancements and error corrections to the code, and possibly develop new interfaces for the C and CLIPS level APIs. These interfaces can be local (graphical or command-line), or yet other Web-based ones.

Next section presents the WebPS structure, and argue why it is better than some previous simulators. Several examples implemented in the WebPS system are described in Section 3. Finally, further development directions are identified.

2 WebPS Software Architecture

The software structure of the simulator has three distinct levels. The inner level is the CLIPS level. This level integrates part of the knowledge about the theoretical description of P systems, the result being a library of CLIPS functions, meta-rules

and templates. Since CLIPS is easily embeddable in C (as its name "C Language Integrated Production System" suggests), we can control the CLIPS level from a C program, and we include some example to illustrate how this is done. The C level is strengthened by introducing a C library for modelling and simulating P systems based on the CLIPS library. The Web application level offers a user-friendly interface to the simulator, and can become, after the addition of debugging and visualization features a powerful P system development tool.

2.1 CLIPS level

This is the core level of the simulator. We address here some crucial issues, namely how we implement in a sequential context the *maximally parallel* and *nondeterministic execution* of P systems.

Maximally parallel execution

The maximally parallel execution requirement relies on constraining our simulation cycle to the following distinct steps:

1. **React** step: where the activated reaction rules are sequentially executed.
2. **Spawn** step: where the new objects created by rules inside their membranes are asserted as object facts; they become visible for a future **React** step.
3. **Communication** step: where the objects injected or ejected by communication rules in different membranes are asserted as objects facts.
4. **Divide** step: it handles possible divisions processes of membranes.
5. **Dissolve** step: it handles dissolving processes of membranes.

By constantly recording the state of the P system after each **React**, **Spawn** and **Communication** steps, we can get a trace of an execution.

Nondeterministic execution

CLIPS uses RETE algorithm to process rules; RETE is a very efficient mechanism for solving many-to-many matching problem [3]. The nondeterministic execution requirement is fulfilled by the CLIPS *random* mechanism. We have looked closely at the random strategy, and we found it makes the same choice for the same configurations in different executions. By calling the random function of CLIPS, the random mechanism uses the random number generator. The failure is related to the improper seeding of the random number generator. We have corrected this error, and properly seed the random number generator by using */dev/urandom*, the entropy gathering device on GNU/Linux systems. This aspect can be also useful for other CLIPS implementations possibly affected by the same failure.

Data representation

An important choice regarding an implementation is given by data representation. We decide to represent P system objects and membrane structure as CLIPS facts (with set-fact-duplication option of CLIPS set to on), and the P systems reaction rules as CLIPS rules. This contrasts the previous implementations which have represented reaction rules as CLIPS facts; while their choice might allow general meta-rules for execution, we get a flexible and efficient framework by representing reaction rules as CLIPS rules. The efficiency is gained by making direct use of the CLIPS pattern-matching mechanism, and rule activation capabilities. This choice is also confirmed by the efficiency of the dissolve and divide operations which imply a lot of moving and copying. Initially we think to represent membranes as modules, but later we see that this representation decreases the efficiency and flexibility of the whole system.

We present the Backus-Naur form of reaction rules, as well as their conversions into CLIPS knowledge-base components:

```
<rule> ::= <object_list> -> [ <output_list> ]
             [ "|" <promoters_inhibitors_list> ]
  <object_list> ::= <object_name> { "+" <object_name> }
  <output_list> ::= <output> \{ "+" <output> }
  <output> ::= <output_object>   |
               <division_marker> |
               <dissolve_marker>
  <output_object> ::= <object_name>
                      [ "(" <target_membrane> ")" ]
  <division_marker> ::= "%"
  <dissolve_marker> ::= "#"
  <promoters_inhibitors> ::= [ <promoter_inhibitor>
                         { "+" <promoter_inhibitor> } ]
  <promoter_inhibitor> ::= <promoter> | <inhibitor>
  <promoter>  ::= <object_name>
  <inhibitor> ::= !<object_name>
  <symport_rule>  ::= "( <object_list>, out | in )"
  <antiport_rule> ::= "( <object_list>, out;
                         <object_list>, in )"
```

It is useful to note that the simulator supports divide, promoters/inhibitors, and symport/antiport rules for membranes. For example, the P system transition rule a+b-¿c+d(2)+e(0) with priority 11 from membrane 1 is converted into the following CLIPS rule:

```
(defrule MAIN::1_a+b->c+d[2]+e[0]
  (declare (salience 11))
  (do (what react))
  (or (parent-child (parent 1) (child 2))
```

```
    (parent-child (parent 2) (child 1)))
(or (parent-child (parent 1) (child 0))
    (parent-child (parent 0) (child 1)))
?a-0 <- (obj (name a) (membrane 1))
?b-1 <- (obj (name b) (membrane 1))
=>
(assert (newobj (name c) (membrane 1)))
(assert (inject (name d) (membrane 2)))
(assert (inject (name e) (membrane 0)))
(retract ?a-0) (retract ?b-1))
```

The membrane structure is reflected by the parent-child facts. An object ***a*** of membrane ***1*** is represented as a CLIPS fact on line 8: *(obj (name a) (membrane 1))*. While the reactants ***a*** and ***b*** are consumed, the new objects ***c***, ***d***, and ***e*** are created during the **Spawn** and **Communication** steps, respectively. The rule priorities are mapped directly to CLIPS rules salience values, and therefore are restricted to integer values in the interval [-10000, 10000].

2.2 C level

In the C level we use CLIPS API, although we plan to develop a complete library that encapsulates the C-CLIPS interface, namely a C library which wrap nicely around the CLIPS one (Figure 1). We represent a P system using XML, and we

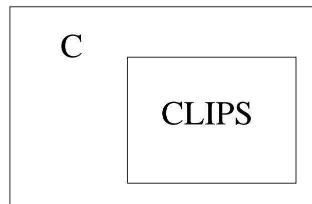

Fig. 1. Relation between the C and CLIPS levels.

define an XML Schema for this kind of document. A special library is developed to handle XML parsing of the input for the CLIPS part of the simulator. In the following example is described a P system conforming to the XML schema. This example system presents a P system for multiplication of the number of ***a*** objects in membrane ***1*** with the number of ***b*** objects in membrane ***0***, the result being the number of ***d*** objects in membrane ***0***.

```
<?xml version="1.0"?>
 <psystem>
   <membrane name="0">
      <object name="b" count="3" />
```

```
      <rule body="b+v->e+v+d" priority="1" />
      <rule body="e+u->b+u+d" priority="1" />
      <rule body="v->u(1)" />
      <rule body="u->v(1)" />
        <membrane name="1">
          <object name="a" count="4" />
          <object name="v" count="1" />
          <rule body="a+v->v(0)" />
          <rule body="a+u->u(0)" />
        </membrane>
    </membrane>
    <query text="count of (objects from 0 where (objects d))" />
  </psystem>
```

2.3 Web level

The Web level of the simulator allows to choose between a user-friendly P system designer written in JavaScript and a traditional HTML input form transmitting an XML description by uploading a file or by editing. Aside from the XML P system description editing, the user can specify a number of executions of the P system. Our JavaScript P system Designer aims to facilitate the description of the P system without requiring the user to write XML, but generating it based on the user's interaction with a dynamic interface. After the user introduces a XML

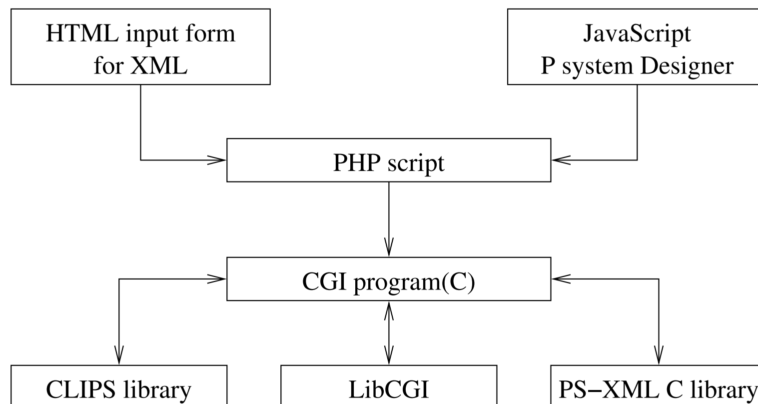

Fig. 2. Structure of the Web level.

description, it is transmitted to a PHP script which does some further processing, and sent then to a CGI program written in C (Figure 2). The C program uses a specific P system XML library called PS-XML, as well as LibCGI and CLIPS

library in order to simulate the evolution of the P system. Finally it returns the results to the user. It is possible to select various information provided as results, and in oprder to help the user to select the desired information we define a query language called PsQL.

PsQL (P systems Query Language)

We define PsQL as an SQL-like language for querying the state of a P system. We developed a CLIPS library for parsing and interpreting this language. At the Web level, the queries can be included in the XML input; these queries are activated after the execution of the specified P system. If it does not exist any query, the P system is simulated, but no output is generated. At the CLIPS level it is possible to specify queries for the P system in a dynamic manner, not just before starting the simulation. At the syntactic level, PsQL is a Lisp-like language, and it is supported by a small CLIPS library of list-handling functions.

The Backus-Naur (partial) description of PsQL is presented in the following lines:

```
<query> ::= <expression>      |
            <count-query>     |
            <membranes-query> |
            <objects-query>
<count-query> ::= "(" count-of <objects-query> |
                               <membranes-query> ")"
<objects-query> ::= "(" objects-from <membranes-spec>
                                [ where <where-spec> ] ")"
<membranes-query> ::= "(" membranes-from <membranes-spec>
                                [ where <where-spec> ] ")"
```

The full description is available at `http://twiki.ieat.ro/twiki/bin/view/Institut/PSystemsQueryLanguage`. We plan to extend PsQL with trace facilities; having queries on the possible traces during an execution represents a step towards an automated verification of the P systems.

3 Examples

Multiplication

The first example is a P system that computes the multiplication of two natural numbers. Figure 3 describes graphically the P system. As inputs we consider the number of $\boldsymbol{a}$ objects in membrane $\boldsymbol{1}$, and the number of $\boldsymbol{b}$ objects in membrane $\boldsymbol{0}$. The result is given by the number of $\boldsymbol{d}$ objects in membrane $\boldsymbol{0}$.

This P system differs from other similar ones by that it does not have exponential space complexity, and does not require active membranes. As a particular

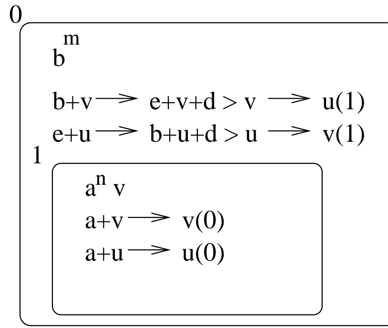

Fig. 3. P system for multiplication.

case, it would be quite easy to compute n^2 by just placing the same number n of ***a*** and ***b*** objects in its membranes.

Another interesting feature is that it may continue computing the multiplication after reaching a certain result. Thus if initially there are m ***b*** objects and n ***a*** objects, the system evolves and reaches a state with $n \cdot m$ ***d*** objects in membrane ***0***. If the user wish to continue in order to compute $(n + k) \cdot m$, it is enough to inject k ***a*** objects in membrane ***1*** at the current state, and the computation can go on. Therefore this example emphasizes a certain degree of re-usability.

Recursive sum

The P system described in Figure 4 computes the recursive sum $\sum_{i=1}^{n} k_i$. The

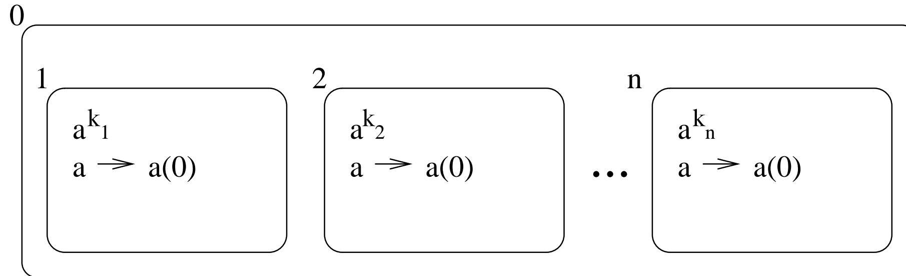

Fig. 4. P system for recursive sum.

numbers of ***a*** objects in the membranes 1...n are the summands, and the result of the computation is the number of ***a*** objects in membrane 0. The PsQL query to determine this result is:

```
(count of (objects from 0))
```

While this example is rather trivial, it illustrates the expressiveness of the query language (PsQL). Using PsQL queries, it is not necessary to apply the rules and execute the specified P system. Given the initial multiset, the same recursive sum is obtained by using the following query:

```
(count of (objects from (membranes from 0)))
```

Dot product of two vectors

Combining the previous two examples, we can compute the dot (scalar) product $x \cdot y$ of two vectors $x, y \in \mathbb{N}^m$, where $m \in \mathbb{N}$. Let us denote the components of the vectors by x_i and y_i, respectively; these components are given by the number of $\boldsymbol{b}$ and $\boldsymbol{a}$ in the membranes labelled by $2i$ and $2i+1$, respectively. Then $x \cdot y$ is given by the number of $\boldsymbol{d}$ objects obtained in membrane 0 after the P system halts. This P system is described in Figure 5: where $k = \overline{0, m-1}$. The PsQL query for

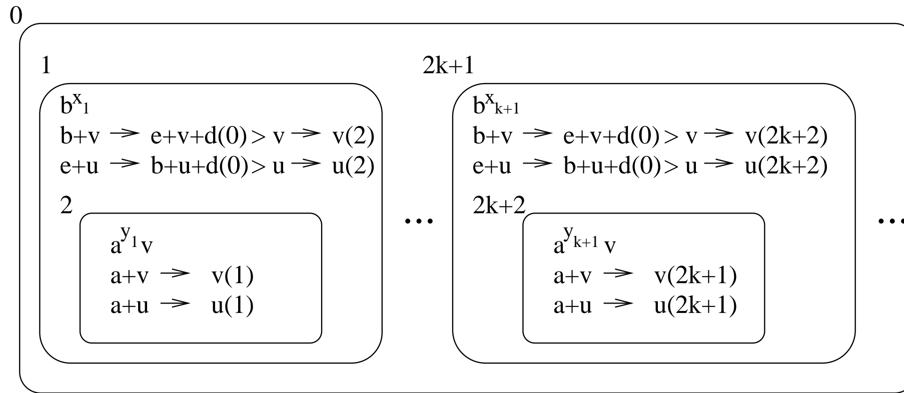

Fig. 5. P system for dot product.

retrieving the result is:

```
(count of (objects from 0))
```

4 Conclusion and Further Work

We present a new simulator of the P systems. It is efficient, flexible, and does not require any previous knowledge in P systems, or expertise in computers. Since the simulator has some novel and interesting features related to efficiency, ease of use and generality, it can become a useful tool for the community, both theoretically and practically. Being GPL licensed, we expect it is eligible to become a simulator benchmark reference. The simulator is available at `http://psystems.ieat.ro`.

We intend to make the simulator available as a web service. Moreover, continuing the commitment to standards compliance, we will strive for SBML compatibility for our specification language. Further improvements are related to better debugging and visualization capabilities (including a flexible fine and coarse-grained tracer), developing a library of macros and methodologies using the principles of modularity, extensibility and structured design from software engineering, introducing flexible rules for the development of macros for P systems.

References

1. G. Ciobanu, W. Guo: P systems running on a cluster of computers. *Proceedings 4th Workshop on Membrane Computing*, LNCS 2933, Springer-Verlag, Berlin, 2004, 123–139.
2. G. Ciobanu, D. Paraschiv: P system software simulator. *Fundamenta Informaticae*, 49, 1-3 (2002), 61–66.
3. C.L. Forgy: RETE. A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence*, 19 (1982), 17–37.
4. Gh. Păun: *Membrane Computing. An Introduction.* Springer-Verlag, Berlin, 2002.
5. M.J. Pérez-Jiménez, F.J. Romero-Campero: A CLIPS simulator for recognizer P systems with active membranes, *Proceedings 2nd Brainstorming Week on Membrane Computing*, University of Sevilla Tech. Rep 01/2004, 387–413.