WEB-PS - A Web-based P System Simulator with Query Facilities

Cosmin Bonchis, Calin Gârboni, Cornel Izbasa, Gabriel Ciobanu[1]

{cbonchis,cgarboni,cizbasa}@info.uvt.ro,gabriel@iit.tuiasi.ro

Abstract: We describe here a tool for simulating P systems [Paun], which we developed since we detected a relative lack of easy, user-friendly and complete software simulators [PSWP]. We used CLIPS (embedded in C) to this end, and we made the simulator available as a web application, complemented by a query language for P systems (PsQL) for specifying the results.
As we found out the idea of using CLIPS to write simulators isn't at all new in the community and there are several noteworthy implementations based on this approach (and some based on other approaches, too).
However, since we believe we can responsibly say that it has some novel and interesting features, some related to efficiency, some related to ease of use and generality, we hope this warrants the simulator as a useful tool for the community [PSWP].

1 Architecture and Functioning

The functionality of the simulator is available at three distinct levels:

1.1 CLIPS Level

This is where all our knowledge about the theoretical simulation of P systems is contained, the result being a library of CLIPS functions, (meta-)rules and templates. C level since CLIPS is easily embeddable in C as its name - "C Language Integrated Production System" - suggests, it is very easy to control the simulator from a C program and we include such example programs, some command-line and a web-based one to illustrate how this is to be done. This level will soon be strengthened by introducing a C library for modelling and simulating P systems based on the CLIPS library. Web application level by this we are trying to offer a most user-friendly interface to the simulator, and which we hope will become, after the addition of more solid debugging and visualization features a true rapid P system development tool. To describe the CLIPS level - the core level of the simulator, we must first address the crucial issue - how we implemented in a sequential context the famed "maximally parallel and nondeterministic execution" of P systems.

1.1.1 Maximally Parallel Execution

To meet the maximally parallel execution requirement we relied on constraining our simulation cycle to these distinct steps:

1. the **React** step - here is where all the reaction rules that are activated are sequentially executed.
2. the **Spawn** step - where all the new created objects created by rules inside their membranes are asserted as object facts - they become visible for future React steps.
3. the **Inject** step - where all the objects injected/ejected by rules in different membranes are asserted as objects facts.
4. the **Divide** step - that handles possible membranes' divisions.
5. the **Dissolve** step - that handles membranes' dissolving processes.

By constantly recording the state of the P system after the first three steps - which can be viewed as a single item for abstraction - and the executed reaction rules, we can get an execution trace.

1.1.2 Nondeterministic Execution

To address the nondeterministic execution requirement we used since the beginning CLIPS' _random_ strategy. However, prof. Ciobanu's questioning of the validity of this choice made us look more closely at the random strategy and found it lacking in a great way - it would always make the same choices for the same rules and facts configurations upon program entry. We suspected this is related to CLIPS' random function and found that the random strategy indeed uses it - by calling the random function we changed the choices for the random strategy - placing the random number generator into a distinct state. This was in turn related to the improper seeding of the RNG, and we addressed this issue by using /dev/urandom, the entropy gathering device on GNU/Linux systems to properly seed it. This may be a solution for other CLIPS implementations that are possibly haunted by the same issue.

1.1.3 Data Representation

The other important choices we had to make were related to data representation. We chose to represent P system objects and membrane structure as CLIPS facts (with CLIPS' set-fact-duplication option set to on) and the reaction rules as CLIPS rules. This contrasts other implementations [3] that have represented reaction rules as CLIPS facts, and while their choice might leave place for more general rules for execution (meta-rules), we think we've managed to realize a sufficiently flexible framework and that we needed the efficiency boost given by representing reaction rules as CLIPS rules, thereby making direct use of the inference engine's pattern-matching and rule activation capabilities. This choice was validated later by the ease of the definition of the "dissolve-with-rules" and "divide" operations, that implied a lot of moving around and copying of rules, which actually meant (re)building them dynamically. Initially we thought about representing membranes as modules, but that seemed to inccur a high efficiency and flexibility penalty, but more study is needed to clearly establish this.

Here is the Backus-Naur Form [BacNau] of reaction rules and their conversions into CLIPS knowledge-base components:

```
<rule> ::= <input_object> { "+" <input_object> } ->
           [ <output> { "+" <output> } ]
  <output> ::= <output_object>   |
               <division_marker> |
               <dissolve_marker>
  <output_object> ::= <object_name>
                      [ "(" <target_membrane> ")" ]
  <division_marker> ::= "%"
  <dissolve_marker> ::= "#"
```

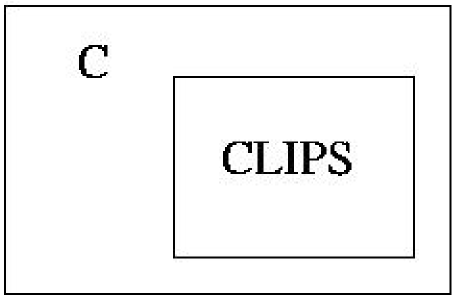
As you can observe, the simulator supports divide and dissolve rules for membranes. For example, the P system rule a+b->c+d(2)+e(0) with priority 11 from membrane 1 is converted into the CLIPS rule:

```
(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. We observe in the rule body that the object a from membrane 1 is represented as the CLIPS fact: (obj (name a) (membrane 1)). Please note that while the a and b reactants are consumed, the c and d, e objects will be created during the Spawn and Inject steps respectively. Also note that the rule priorities are mapped directly to CLIPS rules salience values and therefore are restricted to integer values in the [-10000, 10000] interval.

1.2 The C Level

At this level we now use CLIPS' API, specifically we use CLIPS embedded in the C program, although we plan on developing a complete library that encapsulates this C-CLIPS interface more cleanly, namely a C library that will wrap nicely around the CLIPS one.



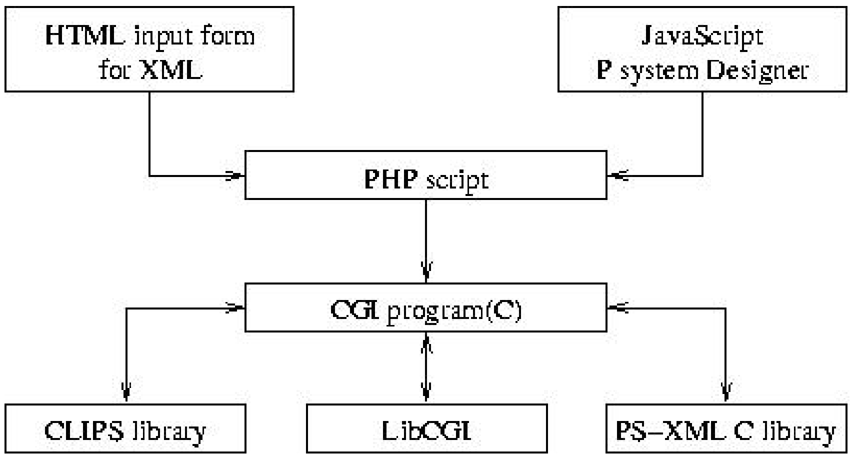
Since we represented P systems using XML - and also defined an XML Schema for this kind of document - the library that we have developed until now handles XML parsing of the input for the CLIPS part of the simulator. Here is an example description of a P system conforming to our schema - this sample system acts as a multiplier for the count of ***a*** objects in membrane 1 and the count 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>
```

For the example command-line and web application programs we have developed we wrote various accessory and output functions for results returned by the CLIPS level. The web application version is presented as a high-level, user-friendly interface to the simulator.

1.3 The Web Level

As a web level user of the simulator, you can choose between a user-friendly P system designer written in JavaScript and a traditional HTML input form that offers the user two ways of transmitting the XML - by uploading a file or by editing the contents of a textarea element. Aside from the XML P system description editing, the user can specify a number of 'runs' of the P system. The JavaScript P system Designer aims to facilitate the description of the P system without requiring the user to write the XML but instead by generating it based on the user's interaction with the dynamic interface.



After the XML is input it is transmitted to a PHP script (that does some further processing) to the CGI program written in C. The C program uses our P system XML library (PS-XML), LibCGI and, of course, the CLIPS library to simulate the evolution of the P system and then return the results to the user. Exactly the problem of deciding what results to return has led us to the idea of defining a query language for P systems - PsQL.

1.3.1 PsQL (P Systems Query Language)

We defined 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 must be specified in the XML input, and after each P system run they are all executed. If you do not input any queries the P system will be simulated but no output will be generated. At the CLIPS level you could specify queries for the P system in a dynamic manner, not just before starting the simulation. At the syntactic level it is a Lisp-like language and this is supported by the development of a small CLIPS library of list-handling functions.
Here is a sample from the Backus-Naur Form for PsQL:

```
<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 in [PsQL].

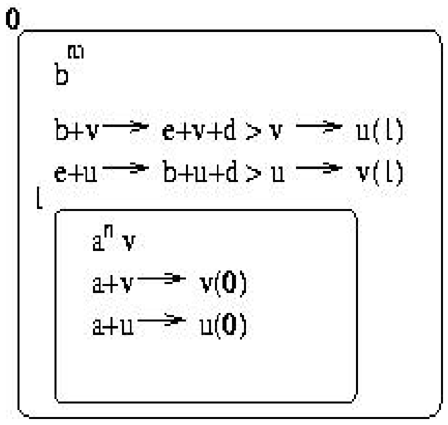
We plan to extend PsQL with trace query facilities. This might aid in the verification of the P systems.

2 Examples

Using our simulator we were able to develop several P systems that implement the basic arithmetic operations.

2.1 Multiplicator P System

The first example is a P system that computes the product of two natural numbers.


0
b^m
$b+v \rightarrow e+v+d > v \rightarrow u(1)$
$e+u \rightarrow b+u+d > u \rightarrow v(1)$
1
$a^n\ v$
$a+v \rightarrow v(0)$
$a+u \rightarrow u(0)$

The inputs to the P system are the number of ***a*** objects in membrane 1 and the number of ***b*** objects in membrane 0.

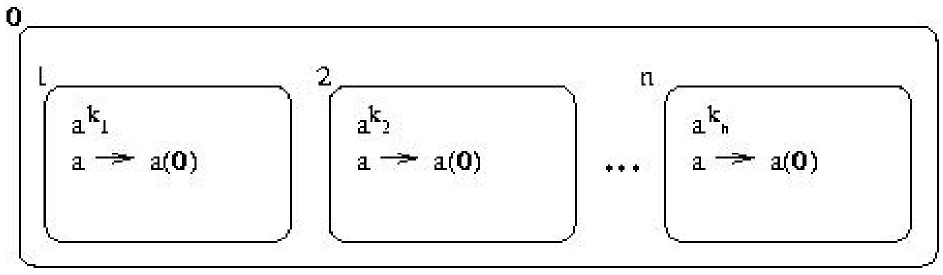
The output is the number of ***d*** objects in membrane 0 and this is equal to $n \quad m$.

This system differs from other similar ones in that it does not have exponential space complexity and does not require active membranes. It would be quite easy to compute n^2 with it by just placing *n* ***a*** and *n* ***b*** objects as inputs.

Another interesting feature is that it can go on computing after reaching a result, namely, if initially there are *m* ***b*** objects and *n* ***a*** objects, the system evolves and reaches a 'final' state with $n \quad m$ ***d*** objects in membrane 0; now, if you also need to compute $(n \quad k) \ m$ you can just inject *k* ***a*** objects in membrane 1 and the computation will go on. So the system manifests a certain degree of reusability.

2.2 P System for the Recursive Sum – Revisited

This P system computes the recursive sum $\,_{i=1}^{n} k_i$.



The 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))
```

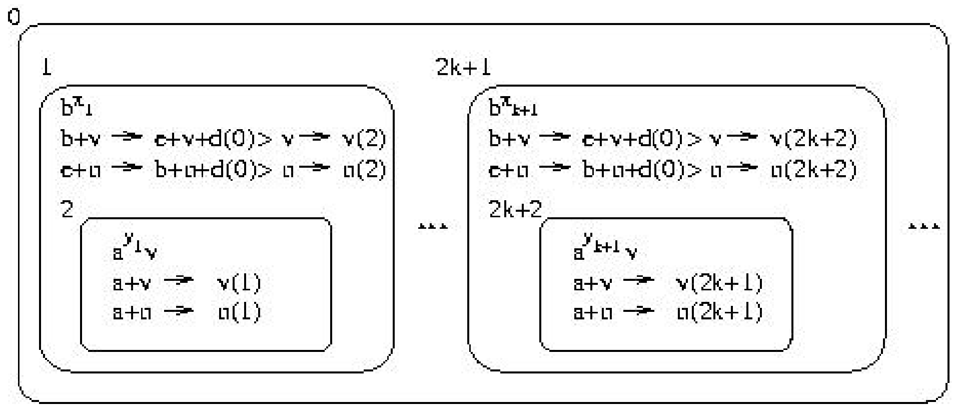
While the example is very well-known and rather trivial, we present it to illustrate the features and expressiveness of our P systems query language - PsQL. The point is, if you can make PsQL queries then you don't even need any reaction rules or actual functioning of the P system. Given the initial multiset without any rules you can obtain the same result with the following query:

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

2.3 P System for the Dot Product of Two Vectors

Combining the previous two examples we can compute the dot product of two vectors, $x \quad y$ with $x, y \quad IN^m$ where *m* *IN.*

Assume the components of the vectors are x_i and y_i respectively. Then, $x \quad y$ is given by the number of ***d*** objects in membrane 0 after this P system has halted.



where $k \in \overline{0...m-1}$.

Since the system is a composition of the recursive sum one and the multiplier, the PsQL query for retrieving the output stays:

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

3 Further Work

- C library for easier modelling of P systems.
- Continuing our commitment to standards compliance, we should be making the simulator available as a web service in the near future.
- On the same note - we will strive for SBML compatibility for our specification language, and this will involve further structuring of our XML "rule" element.
- Better debugging and visualization capabilities (this will include a flexible fine and coarse-grained tracer and very probably support for Sevilla carpets as suggested by prof. Ciobanu).
- Developing a library of macros and methodologies for P systems, which aims to make use of principles of modularity, extensibility and structured design from software engineering for P systems.
- Introducing more flexible rules for aiding in the development of macros for P systems, such as broadcast (send to all children), parentcast (send to parent), broadcast* (send to parent and all children). This would be useful since we would be creating new membranes for which we have no names

beforehand and it would be much easier to specify output to parent with a parentcast for example.

Bibliography

[Paun]

Gh. PAUN Computing With Membranes, October 2000

[WMP]

External link to CDMTCS. Look for Tech. Rep No 140: Pre-proceedings of the Workshop on Multisets Processing, Curtea de Arges (Romania), 2000

[JimCam]

Mario PEREZ-JIMENEZ, Francisco Jose ROMERO-CAMPERO, A CLIPS Simulator for Recognizer P Systems with Active Membranes, Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence University of Sevilla

[WEB-PS]

The P Systems Simulator http://psystems.ieat.ro/

[PSWP]

The P Systems Web Page: http://psytems.disco.unimib.it/

[CioPar]

G.Ciobanu, D.Paraschiv. P-System Software Simulator, Fundamenta Informaticae, vol.49(1-3), 61-66, 2002

[CiPaSt]

G.Ciobanu, Gh.Paun, Gh.Stefanescu. Sevilla Carpets Associated with P Systems. Report 26/03 Rovira i Virgili University, Tarragona, 135-140, 2003

[BacNau]

ISO/IEC 14977:1996(E) document

[W3C]

XML Schema Part 1: Structures Second Edition W3C Recommendation 28 October 2004 http://www.w3.org/TR/xmlschema-1/

[PsQL]

Backus-Naur Form for PsQL
http://twiki.ieat.ro/twiki/bin/view/Institut/PSystemsQueryLanguage