Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 171 (2007) 171–186
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Encoding Catalytic P Systems in π@

Cristian Versari[1]

Dipartimento di Scienze dell'Informazione
Universita' di Bologna
Bologna, Italy

Abstract

P systems are theoretical computing devices abstracted away from the biological architecture of the cell, introduced some years ago by Gheorghe Păun and now intensely studied. In the area of concurrent systems, process calculi have recently been applied and extended with similar aim, to simulate (and formalise) the behaviour of the cell. Although many common points can be found between the two approaches, no formal and exhaustive comparison has been carried out yet.
π@ is a new calculus, strongly π-Calculus based, well-suited to easily encode biologically inspired process calculi. In this paper the encoding in π@ of one variant of P systems is proposed, thus allowing a better understanding of similarities between P systems and bio-inspired process calculi.

Keywords: systems biology, membrane computing, catalytic P systems, process algebra, priority

1 Introduction

Membrane computing ([6]) arises as a branch of natural computing, with the aim of formalising and studying biologically inspired devices — P systems — from the point of view of automata and formal languages theory. Though initially they were not intended to provide models for the cell, recent research reported many advances in this direction, thus meeting the efforts of recent applications of another computer science research area — concurrency theory — to biology. In the first of these applications ([10]) biological entities (molecules, enzymes, and so on) were modeled in π-Calculus ([4,5]) as processes and pathways as sequences of inter-process communications. While very suited to model mobility and interaction of biological elements, π-Calculus lacked an explicit expression for the idea of *compartment*. Subsequent approaches (like [9,8,2]) added that notion, trying to retain concurrency and mobility capabilities: compartment becomes an ambient in BioAmbients, a box in Beta Binders, or the content of a membrane in Brane Calculus. Although underlying different sides of the biology of the cell, these languages presented many

[1] Email: versari@cs.unibo.it

doi:10.1016/j.entcs.2007.05.015

common aspects due to this idea of compartment: localisation of interaction within the same compartment, atomicity of complex operations, mass object movements between different compartments. Even if P systems come from a very different computer science branch, they present many similar aspects to these process calculi — concurrency/distribution, nondeterminism, interaction, mobility, compartmentalisation — which make a comparison very interesting. One way to perform this comparison is to encode all of them in the same language.

π@ is a conservative, strongly π-Calculus based language with two key features: polyadic synchronisation (structured names for channels) to represent compartments, and priority to achieve atomicity for complex, mass movement operations. These features allowed simple encodings in π@ of biologically inspired calculi like Bioambients and Brane Calculi ([11]), with the aim of showing their common characteristics. To this same purpose we now present the encoding of a simple kind of P systems, by exploiting the key features of π@ to overcome the significant differences between membrane systems and process algebras (first of all maximal parallelism), thus proposing π@ as a new core calculus suited to encode and compare a wide variety of biologically inspired formalisms.

This paper is organised as follows: in section 2 basic definitions for catalytic P systems are recalled, then in section 3 π@ is introduced, with some example; section 4 presents step by step the complete definition for an encoding function in π@ of catalytic P systems. In section 5 some final observation is posed.

2 P systems

Quoting from [7]:

> "The essential ingredient of a P system is its membrane structure, which can be a hierarchical arrangement of membranes, like in a cell (hence described by a tree), or a net of membranes (placed in the nodes of a graph), like in a tissue, or in a neural net. The intuition behind the notion of a membrane is that from biology, of a three-dimensional vesicle, but the concept itself is generalised/idealised to interpreting a membrane as a separator of two regions (of the Euclidean space), a finite *inside* and an infinite *outside*, also providing the possibility of a selective communication among the two regions.
>
> The variety of suggestions from biology and the range of possibilities to define the architecture and the functioning of a membrane-based-multiset-processing device are practically endless – and already the literature of membrane computing contains a very large number of models."

The high number of P systems models imposes a choice on the most similar to process calculi and initially easier to encode: catalytic P systems have been chosen for the simplicity of evolution rules and static structure of the membrane tree. Below, preliminary definitions are recalled, then formal definition of catalytic P systems is given.

Definition 2.1 Given a set S, a *finite multiset* over S is a function $m : S \rightarrow \mathbb{N}$ such that the set $dom(m) = \{s \in S \,|\, m(s) \neq 0\}$ is finite. The *multiplicity* of an element s in m is given by the natural number $m(s)$. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. A multiset m such that $dom(m) = \emptyset$ is called *empty* . The empty multiset is denoted by $\emptyset$.

Given the multiset m and m', $m \subseteq m'$ holds if $m(s) \leq m'(s)$ for all $s \in S$ while $\oplus$ denotes their *multiset union*: $m \oplus m'(s) = m(s) + m'(s)$. The operator $\setminus$ denotes *multiset difference*: $(m \setminus m')(s) =$ if $m(s) \geq m'(s)$ then $m(s) - m'(s)$ else 0. The *scalar product*, $j \cdot m$, of a number j with m is $(j \cdot m)(s) = j \cdot (m(s))$. The cardinality of a multiset is the number of occurrences of elements contained in the multiset: $|m| = \sum_{s \in S} m(s)$.

Some basic definitions on strings, cartesian products and relations are then given.

Definition 2.2 A string over S is a finite (possibly empty) sequence of elements in S. The length of a string is the number of occurrences of elements contained in S. With S^* we denote the set of strings over S, and $u, v, w, \ldots$ range over S.

Given a string $u = x_1 \ldots x_n$, the multiset corresponding to u is defined as follows: for all $s \in S$, $m_u(s) = |\{i \mid x_i = s \wedge 1 \leq i \leq n\}|$. With abuse of notation, we use u to denote also m_u.

The definition of membrane structure follows.

Definition 2.3 Given the alphabet $V = \{[,]\}$, the set MS is the least set inductively defined by the following rules:

- $[\,] \in MS$
- if $\mu_1, \mu_2, \ldots, \mu_n \in MS$, $n \geq 1$, then $[\mu_1 \ldots \mu_n] \in MS$

We define the following relation over MS: $x \sim y$ if and only if the two strings can be written in the form $x = [_1 \ldots [_2 \ldots]_2 \ldots [_3 \ldots]_3 \ldots]_1$ and $y = [_1 \ldots [_3 \ldots]_3 \ldots [_2 \ldots]_2 \ldots]_1$ (i.e., if two pairs of parenthesis that are neighbours can be swapped together with their contents).

The set $\overline{MS}$ of *membrane structures* is defined as the set of equivalence classes w.r.t. the relation $\sim^*$.

We call a *membrane* each matching pair of parenthesis appearing in the membrane structure. A membrane structure μ can be represented as a Venn diagram, in which any closed space (delimited by a membrane and by the membranes immediately inside) is called a *region* of μ.

Definition 2.4 A *catalytic P system* (of degree d, with $d \geq 1$) is a construct

$$\Pi = (V, C, \mu, w_1^0, \ldots, w_d^0, R_1, \ldots, R_d, i_0)$$

where

(i) V is a finite alphabet whose elements are called *objects*;

(ii) $C \subseteq V$ is a set of *catalysts*;

(iii) μ is a *membrane structure* consisting of d membranes (usually labelled with i and represented by corresponding brackets $[_i$ and $]_i$, with $1 \leq i \leq d$);

(iv) w_i^0, $1 \leq i \leq d$, are strings over V associated with the regions $1, 2, \ldots, d$ of μ; they represent multisets of objects present in the regions of μ (the multiplicity of a symbol in a region is given by the number of occurrences of this symbol in the string corresponding to that region);

(v) R_i, $1 \leq i \leq d$, are finite sets of *evolution rules* over V associated with the regions $1, 2, \ldots, d$ of μ; these evolution rules are of the forms $a \to v$ or $ca \to cv$, where c is a catalyst, a is an object from $V \setminus C$, and v is a string from $((V \setminus C) \times \{here, out, in\})^*$;

(vi) i_0 is a number between 1 and d and it specifies the *output* membrane of Π.

3 The π@ language

The π@ calculus — pronounced like the french "paillette" — is essentially π-Calculus with the addition of two features: polyadic synchronisation and prioritised communication.

Polyadic Synchronisation

In π-Calculus channels and names are usually synonyms. Polyadic synchronisation ([1]) introduces a sort of *structure* in channels: they are represented by a *vector* of one or more names. The idea of compartment is then expressed by adopting for communication a channel composed of two parts. One common example is the representation of an e-mail address: *user@domain* is a channel composed of 2 names, both necessary to send an email to the desired user. Polyadic syntax is represented in π@ exactly in the same way: names in a channel are separated by the character '@'. Thus, the communication of the datum d on the channel name *chan* in the compartment *comp* is usually written in π@ as

$$\overline{chan@comp}\langle d\rangle.P \;\big|\; chan@comp(x).Q \quad \to \quad P \;\big|\; Q\{d/x\}$$

where the first process sends d and then becomes the process P, while the second receives d and executes the actions specified in Q. In π@ semantics, interaction between two processes may occur only if the channels involved in the input/output operation are composed of the same number of names, with the same multiplicity and in the same order. Consequently, we have

$$\overline{x@y}\langle\rangle.P \;\big|\; y@x().Q \quad \not\to \qquad\qquad (x \neq y)$$

$$\overline{x}\langle\rangle.P \;\big|\; x@x().Q \quad \not\to$$

because the channel $x@y$ is different from $y@x$, and x is different from $x@x$.

Priority

Priority behaves as expected: a high-priority process holds the central processing unit and executes its job before any low priority process. In π@ high priority synchronisations or communications are executed before any other low priority action. Usually a high priority action is indicated by underlining the name of the channel one or more times. For example, the expression

$$\overline{stand}\langle x\rangle.P \mid \overline{\underline{walk}}\langle y\rangle.Q \mid \overline{\underline{\underline{run}}}\langle z\rangle$$

contains three processes with different, increasing priority. To express more than three levels of priority another notation is used, where the priority of the process is represented by a number following the channel names. The above expression may be rewritten as

$$\overline{stand:2}\langle x\rangle.P \mid \overline{walk:1}\langle y\rangle.Q \mid \overline{run:0}\langle z\rangle$$

where a lower priority action is labelled with a higher number (the highest priority is denoted by 0).

Interaction between processes may occur only if channels have the same priority. In this example

$$\overline{\underline{x}}\langle y\rangle.P \mid x(z).Q \quad \not\rightarrow$$

$$\overline{\underline{x}}\langle y\rangle.P \mid \underline{x}(z).Q \quad \rightarrow \quad P \mid Q\{y/z\}$$

only the second interaction is allowed, because the expressions x and $\underline{x}$ denote actually two different channels. Finally, as expected, low priority actions occur only if no higher priority action may occur:

$$\begin{array}{l} \bar{l}\langle w\rangle \mid l(x).P \mid \overline{\underline{h}}\langle y\rangle \mid \underline{h}(z).Q \quad \not\rightarrow \\ \mathbf{0} \mid P\{w/x\} \mid \overline{\underline{h}}\langle y\rangle \mid \underline{h}(z).Q \end{array}$$

$$\begin{array}{l} \bar{l}\langle w\rangle \mid l(x).P \mid \overline{\underline{h}}\langle y\rangle \mid \underline{h}(z).Q \quad \rightarrow \\ \bar{l}\langle w\rangle \mid l(x).P \mid \mathbf{0} \mid Q\{y/z\} \quad \rightarrow \\ \mathbf{0} \mid P\{w/x\} \mid \mathbf{0} \mid Q\{y/z\} \end{array}$$

Interactions on low-priority channel l may happen only after the high-priority communication on channel $\underline{h}$.

In the next section the syntax and reduction semantics of π-Calculus is briefly recalled, then in 3.2 the simple extensions which lead to π@ are presented.

3.1 *π-Calculus*

Definition 3.1 Let

$\mathcal{N}$ be a set of names on a finite alphabet, $x, y, z, \ldots \in \mathcal{N}$;

$\overline{\mathcal{N}} = \{\overline{x} \mid x \in \mathcal{N}\}$

The syntax of π-Calculus is defined as

$$\begin{array}{lcl} P & ::= & \mathbf{0} \;\Big|\; \sum_{i\in I} \pi_i.P_i \;\Big|\; P \mid Q \;\Big|\; !\,P \;\Big|\; (\nu x)P \\ \pi & ::= & \tau \;\Big|\; x(y) \;\Big|\; \overline{x}\langle y\rangle \end{array}$$

Definition 3.2 The congruence relation $\equiv$ is defined as the least congruence satisfying alpha conversion, the commutative monoidal laws with respect to both ($\mid$,$\mathbf{0}$) and (+,$\mathbf{0}$) and the following axioms:

$$\begin{array}{rcll} (\nu x)P \mid Q & \equiv & (\nu x)(P \mid Q) & \text{if } x \notin fn(Q); \\ (\nu x)P & \equiv & P & \text{if } x \notin fn(P) \\ !\,P & \equiv & !\,P \mid P & \end{array}$$

where the function fn is defined as

$$\begin{array}{lcl} fn(\tau) & \overset{def}{=} & \emptyset \\ fn(x(y)) & \overset{def}{=} & \{x\} \\ fn(\overline{x}\langle y\rangle) & \overset{def}{=} & \{x, y\} \\ fn(\mathbf{0}) & \overset{def}{=} & \emptyset \\ fn(\pi.P) & \overset{def}{=} & fn(\pi) \cup fn(P) \\ fn(\sum_{i\in I} \pi_i.P_i) & \overset{def}{=} & \bigcup_i fn(\pi_i.P_i) \\ fn(P \mid Q) & \overset{def}{=} & fn(P) \cup fn(Q) \\ fn(!\,P) & \overset{def}{=} & fn(P) \\ fn((\nu x)P) & \overset{def}{=} & fn(P) \setminus \{x\} \end{array}$$

Definition 3.3 π-Calculus semantics is given in terms of the reduction system described by the following rules:

$$\frac{}{\tau.P \;\to\; P} \qquad \frac{P \;\to\; P'}{(\nu\, x)P \;\to\; (\nu\, x)P'}$$

$$\frac{}{(\mu(y).P + M) \mid (\overline{\mu}\langle z\rangle.Q + N) \;\to\; P\{z/y\} \mid Q}$$

$$\frac{P \;\to\; P'}{P \mid Q \;\to\; P' \mid Q} \qquad \frac{P \equiv Q \quad P \;\to\; P' \quad P' \equiv Q'}{Q \;\to\; Q'}$$

See [4] for an exaustive introduction to π-Calculus.

3.2 π@ syntax and semantics

π@ is very close to *pi*-Calculus: from a syntactical point of view the only difference is the structure of channels, composed of multiple names followed by the priority of the action. We use μ to denote a vector of names $x_1, \ldots, x_n$ and $\mu : k$ to denote a channel, that is a vector of names μ followed by a colon and a natural number k specifying the priority. As usual, $\overline{\mu : k}$ represents an output operation along channel $\mu : k$, while $\alpha : k$ stands for a generic input, output or silent action τ of priority k.

Definition 3.4 Let

$\mathcal{N}$ be a set of names on finite alphabet, $x, y, z, \ldots \in \mathcal{N}$;
$\mathcal{N}^+ = \bigcup_{i>0} \mathcal{N}^i$, $\quad \mu \in \mathcal{N}^+$;
$\overline{\mathcal{N}}^+ = \{\overline{\mu} \mid \mu \in \mathcal{N}^+\}$;
$\alpha \in \left(\overline{\mathcal{N}}^+ \cup \mathcal{N}^+ \cup \{\tau\} \right)$;

The syntax of π@ defined as

$$\begin{array}{lcl} P & ::= & \mathbf{0} \;\Big|\; \sum_{i\in I} \pi_i.P_i \;\Big|\; P \mid Q \;\Big|\; !\,P \;\Big|\; (\nu x)P \\ \pi & ::= & \tau{:}k \;\Big|\; \mu{:}k(x) \;\Big|\; \overline{\mu{:}k}\langle x\rangle \end{array}$$

As previously introduced, some abbreviations are very often used in this paper:

$$\begin{array}{lcllcl} \mu(x) & = & \mu{:}2(x) & \qquad \overline{\mu}\langle x\rangle & = & \overline{\mu{:}2}\langle x\rangle \\ \underline{\mu}(x) & = & \mu{:}1(x) & \qquad \underline{\overline{\mu}}\langle x\rangle & = & \overline{\mu{:}1}\langle x\rangle \\ \underline{\underline{\mu}}(x) & = & \mu{:}0(x) & \qquad \underline{\underline{\overline{\mu}}}\langle x\rangle & = & \overline{\mu{:}0}\langle x\rangle \end{array}$$

The definition for structural congruence $\equiv$ is exactly the same as given for π-Calculus, where the function fn is naturally extended to the π@ syntax. The reduction semantics is very similar, but defined in terms of an auxiliary function $I^k(P)$, representing the set of actions of priority k which the process P may immediately execute. For example, if

$$P = a.Q \mid \underline{b} \mid \underline{\underline{\overline{c}}}.R \mid \underline{\overline{d}} + \underline{\underline{e}}.S \mid \overline{a}.T$$

then $I^0(P) = \{\overline{c}, e\}$, $I^1(P) = \{b, \overline{d}\}$, $I^2(P) = \{a, \overline{a}, \tau\}$, where the availability of τ action derives from the interaction of the first and last process.

Definition 3.5 Let $I^k(P)$ be

$$\begin{aligned}
I^k\big(\sum_i \alpha_i\!:\!l_i.P_i\big) &= \{\alpha_i \mid l_i = k\};\\
I^k\big((\nu\ y)\ P\big) &= I^k(P) \setminus \big\{\alpha \mid y \in \{x_1,\ldots,x_n\} \wedge \\
&\qquad (\alpha = x_1@\ldots@x_n\ \vee\ \alpha = \overline{x_1@\ldots@x_n})\big\};\\
I^k\big(!P\big) &= I^k(P \mid P);\\
I^k\big(P \mid Q\big) &= I^k(P) \cup I^k(Q) \cup \{\tau \mid I^k(P) \cap \overline{I^k(Q)} \neq \emptyset\},\\
&\qquad with \overline{I^k(Q)} = \big\{\overline{\alpha} \mid \alpha \in I^k(Q)\big\}
\end{aligned}$$

$\pi@$ semantics is given in terms of the following reduction system:

$$\frac{\tau \notin \bigcup_{i<k} I^i(M)}{\tau\!:\!k.P + M\ \rightarrow_k\ P} \qquad \frac{P\ \rightarrow_k\ P'}{(\nu\ x)P\ \rightarrow_k\ (\nu\ x)P'}$$

$$\frac{\tau \notin \bigcup_{i<k} I^i(M \mid N)}{(\mu\!:\!k(y).P + M) \mid (\overline{\mu}\!:\!k\langle z\rangle.Q + N)\ \rightarrow_k\ P\{z/y\} \mid Q}$$

$$\frac{cP\ \rightarrow_k\ P' \qquad \tau \notin \bigcup_{i<k} I^i(P \mid Q)}{P \mid Q\ \rightarrow_k\ P' \mid Q} \qquad \frac{P \equiv Q \quad P\ \rightarrow_k\ P' \quad P' \equiv Q'}{Q\ \rightarrow_k\ Q'}$$

$\pi@$ reduction rules are exactly the same of π-Calculus, except for the additional condition $\tau \notin \bigcup_{i<k} I^i(\ldots)$ which avoids the execution of low priority actions if higher priority communications (represented by τ actions) are immediately available.

For a comprehensive formal treatment of priority in process algebras, see [3].

4 Encoding catalytic P systems in $\pi@$

P systems and process calculi present many similar aspects: interaction between different elements (by direct communication in π-Calculus-like languages, by transition rules in P systems), localisation of interaction (within the same membrane in P systems, within the scope of a name in π-Calculus-like languages or within the same compartment in bio-inspired calculi), concurrency, nondeterminism. Anyway, they come from distant areas, in fact they are composed of totally different elements: P systems are made of objects, (sometimes prioritised) rules, membranes. In process calculi there is only one kind of elements: processes. So it is quite natural expecting that every element of a P system will be translated to a $\pi@$ process. One non trivial difference to overcome is the maximal parallelism typical of P systems: even if process calculi deal with concurrent objects and describe parallel evolution, there is no immediate correspondence with such a strong constraint in $\pi@$, which means that maximal parallelism must be injected in some way within the behaviour of every process.

In the next section basic ideas of encoding catalytic P systems in $\pi@$ are presented, first defining the general form of the encoding function and then specifying its simplest parts without considering the difficulties introduced by maximal parallelism. In section 4.2 the encoding function is finally specified, preserving modularity and divergence/termination properties of the encoded systems even after the introduction of maximal parallelism constraints.

4.1 Encoding ideas

Obviously there is no unique translation. The intention is to choose the one which best fits some characteristics we require.

First, the encoding should preserve some kind of *operational correspondence*: in other words, if we have a transition between two configurations of a P system $C_1 \longrightarrow C_2$, the encoded system $[\![C_1]\!]$ should be able to perform (in one or more steps) a transition $[\![C_1]\!] \Longrightarrow [\![C_2]\!]$, while if $[\![C_1]\!] \Longrightarrow Q$, then a configuration C should exist such that $Q \Longrightarrow [\![C]\!]$ and $C_1 \longrightarrow \ldots \longrightarrow C$.

Another important requirement is *modularity*: the encoding of an element shall be independent of the encoding of any other element, disregarding the order which they have been taken into account, the membrane they are going to be located in, or the number and type of the adjacent elements. In other words, we want to be able to translate a P system, analyse or execute it, and later to change it by adding (or eliminating) objects or rules (or even membranes with arbitrary content) without having to re-encode all the system, or better without modifying at all the already encoded part.

Finally, a reasonable condition is the preservation of *divergence / termination properties*: it is advisable that the encoded systems terminate their computation if (and only if) the original ones do, but this raises many problems because of the maximal parallelism typical of P systems.

In this section we pursue the first two requirements, trying to make clear the general encoding ideas, while in the next section we show the final version of the encoding function which also preserves divergence/termination properties.

Before sketching the first encodings, we define the encoding function in its general form:

Definition 4.1 Given a catalytic P system $\Pi \in \Pi_{Dom}$

$$\Pi = (V, C, \mu, w_1^0, \ldots, w_d^0, R_1, \ldots, R_d, i_0)$$

where Π_{Dom} represents the domain of all the possible catalytic P systems, the encoding function $[\![\ ]\!] : \Pi_{Dom} \longrightarrow \mathcal{P}$ is defined as

$$[\![\Pi]\!] \overset{def}{=} [\![\mu]\!]_{MS} \mid [\![w_1^0, 1]\!]_S \mid \ \ldots \ \mid [\![w_d^0, d]\!]_S \mid$$
$$[\![R_1, 1]\!]_R \mid \ \ldots \ \mid [\![R_d, d]\!]_R$$

where

$$[\![\ ]\!]_{MS} : \overline{MS} \longrightarrow \mathcal{P}, \qquad [\![\ ]\!]_{S} : V^{*} \times I\!N \longrightarrow \mathcal{P}, \qquad [\![\ ]\!]_{R} : R_{Dom} \times I\!N \longrightarrow \mathcal{P},$$

$\overline{MS}$, V^{*}, R_{Dom} standing respectively for the sets of all possible membrane structures, multisets of objects (i.e. strings) and evolution rules.

The next task is the definition of $[\![\ ]\!]_{MS}$, $[\![\ ]\!]_{S}$, $[\![\ ]\!]_{R}$, on which the definition of $[\![\ ]\!]$ depends, but some remarks are due before.

Even without specifying the functions $[\![\ ]\!]_{MS}$, $[\![\ ]\!]_{S}$, $[\![\ ]\!]_{R}$, it is easy to observe that the encoding is completely modular: every piece can be added, eliminated or modified without affecting the rest of the encoded system. In this way we obtain exactly our aim, rules do not depend on objects and viceversa, but there is something more: objects or rules encoding does not depend on membrane structure, since the only information needed is the number (or the label) of the membrane containing the object or rule.

A P systems is essentially a set of objects observing certain evolution rules: if we have the rule $ca \to cbd$, the object a can evolve to two new objects b, d, but only if a catalyst c is nearby. Perhaps the most natural way to compose the steps of this evolution is: "if an object of type a meets the catalyst c, then it evolves into two new objects b and d", thus the objects are thought as the main actors of this process. But if we add another rule, $a \to e$, then the behaviour of the object a considerably changes: "one possibility is that if the object a meets the catalyst c, then it evolves into two new objects b and d; another one is that the object a evolves into the object e". So if we think objects as the main actors of evolution, probably we are not going to find a modular encoding. The key for avoiding this problem is to spot the real actors: evolution rules. The rule $ca \to cbd$ could be expressed in this way: "forever do: check if an object of type a is nearby, then check if a catalyst c is nearby too; if they are, replace a with b and d". Hence, adding another rule would be not difficult: $a \to e$ would just become another process executing "forever do: if a is nearby, then replace it with e". The behaviour of objects would then become trivial, in fact a would be "if some rule asks for object a, say a is present". Obviously other issues would then pop up, caused for example by rules competing for the same objects, but we will face them later.

Following this interpretation, we could have (ignoring the second parameter of the encoding functions, for the moment)

$$[\![\, a \,]\!]_{S} = \overline{a}\,, \qquad [\![\, c \,]\!]_{S} = \overline{c}$$

$$[\![\, ca \to cbd \,]\!]_{R} \quad = \quad !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d})$$

In fact

$$\begin{array}{l}\overline{a} \mid \overline{c} \mid !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d}) \quad \rightarrow \\ \overline{a} \mid a.(\overline{c} \mid \overline{b} \mid \overline{d}) \mid !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d}) \quad \rightarrow \\ \overline{c} \mid \overline{b} \mid \overline{d} \mid !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d})\end{array}$$

but in this way if the object a is not present, object c disappears even if not used and the rule process is deadlocked:

$$\overline{c} \mid !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d}) \quad \rightarrow \quad a.(\overline{c} \mid \overline{b} \mid \overline{d}) \mid !\, c.a.(\overline{c} \mid \overline{b} \mid \overline{d})$$

So a correction is necessary

$$[\![\, ca \rightarrow cbd \,]\!]_R \quad = \quad !\, c.(\overline{c} + a.(\overline{c} \mid \overline{b} \mid \overline{d}))$$

so that the checking phase does not delete even unused objects. But another issue appears, because the rule above may execute forever without producing its output objects, even if input objects a and c are present: we have introduced divergence. In fact, after checking the presence of c, we should be able to give some kind of precedence to the second term of the choice, in order to ensure that if some object a is present, then the rule completes its job. By means of prioritised choice, it is possible to model this kind of precedence:

$$[\![\, ca \rightarrow cbd \,]\!]_R \quad = \quad !\, \underline{c}.(\tau.\underline{\overline{c}} + \underline{a}.(\underline{\overline{c}} \mid \underline{\overline{b}} \mid \underline{\overline{d}}))$$

In this way we obtain that if a given rule *may* be applied, for sure it is applied, anyway divergence is not completely removed yet: if only object c is present, the above process would continue its checking forever. Since maximal parallelism introduces again the same kind of problem, we defer the final solution to the next section.

One aspect we can consider before dealing with complex encodings is the localisation of communication: objects located in one membrane shall not interact with rules located in another membrane. $\pi@$ allows to model this constraint in a very simple way, by means of polyadic synchronisation. The encoding functions $[\![\ ]\!]_R$, $[\![\ ]\!]_S$ require two parameters: the first is the object or rule to be encoded, the second is the number (or label) of the surrounding membrane. Localised communication is then obtained easily by specifying the membrane number for every object manipulated by evolution rules:

$$[\![\, c, n \,]\!]_R \quad = \quad \underline{\overline{c@n}}\,, \qquad [\![\, a, n \,]\!]_R \quad = \quad \underline{\overline{a@n}}$$

$$[\![\, ca \rightarrow cbd, n \,]\!]_R \quad = \quad !\, \underline{c@n}.(\tau.\underline{\overline{c@n}} + \underline{a@n}.(\underline{\overline{c@n}} \mid \underline{\overline{b@n}} \mid \underline{\overline{d@n}}))$$

The next step is to understand how to express rules producing objects outside membrane boundaries. For example, encoding the rule $ca \rightarrow c(b, out)$, which ejects b in the external membrane, requires some knowledge about membrane structure.

The definition 4.1 asserts that the only process owing this knowledge is $[\![\ \mu\ ]\!]_{MS}$, which is supposed to be properly queried. Thus, a possible solution may be

$$[\![\ ca \rightarrow c(b, out), n\ ]\!]_R \quad = \quad !\ \underline{c@n}.(\tau.\overline{c@n} + \underline{a@n}.(\overline{c@n} \mid \underline{out@n}(x).\overline{b@x}))$$

where the instruction $\underline{out@n}(x)$ queries the membrane structure process about the label of the outer membrane.

Hence, membrane structure process could be seen as a service replying to two types of queries: "tell me the label of the membrane surrounding membrane n", or "tell me the label of one of membranes inside membrane n". The second type of query introduces some kind of nondeterminism, because each membrane may contain more than one child membrane, so it is necessary to preserve this nondeterministic behaviour after the encoding. The easiest way to translate a membrane structure in $\pi@$ is considering each membrane as a separated process. Thus, the first type of query imposes that every membrane process "knows" the label of its surrounding membrane. In other words, if membrane structure is a tree, we are going to implement it with *pointers* from child membranes to parent membranes. But the second type of rule requires that pointers from parent membranes to the inner ones also exist: including these pointers in the child membranes processes allows to encode each membrane completely disregarding its content.

Definition 4.2 Given a membrane structure $\mu \in \overline{MS}$, the encoding function

$$[\![\ ]\!]_{MS} : \overline{MS} \longrightarrow \mathcal{P}$$

is defined as

$$[\![\ [_1\ \mu_{i_1}, \ldots, \mu_{i_k}\ ]_1\ ]\!]_{MS} \quad \overset{def}{=}$$

$$!\ \overline{out@1}\langle outside\rangle \mid [\![\ \mu_{i_1}, 1\ ]\!]'_{MS} \mid \ldots \mid [\![\ \mu_{i_k}, 1\ ]\!]'_{MS}$$

where

$$[\![\ ]\!]'_{MS} : \overline{MS} \times I\!N \longrightarrow \mathcal{P}$$

is defined as

$$[\![\ [_n\ \mu_{i_1}, \ldots, \mu_{i_k}\ ]_n,\ p\ ]\!]'_{MS} \quad \overset{def}{=}$$

$$!\ \overline{out@n}\langle p\rangle \mid !\ \overline{in@p}\langle n\rangle \mid [\![\ \mu_{i_1}, n\ ]\!]'_{MS} \mid \ldots \mid [\![\ \mu_{i_k}, n\ ]\!]'_{MS}$$

The main encoding function $[\![\ ]\!]_{MS}$ is defined in terms of an auxiliary function $[\![\ ]\!]'_{MS}$, which requires an additional parameter: the label of the parent membrane. The skin has no parent membrane, so we provide a fictitious name *outside* to express the compartment containing the objects ejected from skin and never allowed to enter again. The encoding of each membrane is completely modular, hence any change to the membrane structure tree affects only the encodings of the nodes directly involved: adding or removing a whole subtree requires no changes to the remaining structure.

Before facing divergence issues and maximal parallelism, the definition of $[\![\ ]\!]_S$ is given.

Definition 4.3 Given a string $s \in V^*, s = s_1 s_2 \dots s_n$, the encoding function

$$[\![\ ]\!]_S : V^* \longrightarrow \mathcal{P}$$

is defined as

$$[\![\ s\ ]\!]_S \overset{def}{=} \overline{\underline{\underline{s_1}}} \mid \overline{\underline{\underline{s_2}}} \mid \dots \mid \overline{\underline{\underline{s_n}}}$$

Again, the encoding is completely modular. Furthermore, it does not distinguish catalysts from common objects, in fact the distinction is only useful before compile-time to check the correctness of P system rules. The reason for the higher degree of priority will be clear in the next section.

4.2 Final encodings

Two issues persist: the divergence introduced in the encoding of evolution rules and maximal parallelism. Divergence is caused essentially by an endless loop. In fact, the rule

$$[\![\ ca \rightarrow cbd\ ]\!]_R = !\, \underline{c}.(\tau.\overline{\underline{c}} + \underline{a}.(\overline{\underline{c}} \mid \overline{\underline{b}} \mid \overline{\underline{d}}))$$

contains a guarded loop, but the guard has no effect if the catalyst c is present, because it never disappears. So an additional guard is required:

$$[\![\ ca \rightarrow cbd\ ]\!]_R = \overline{coin} \mid !\, coin.\underline{c}.(\tau.\overline{\underline{c}} + \underline{a}.(\overline{\underline{c}} \mid \overline{\underline{b}} \mid \overline{\underline{d}} \mid \overline{coin}))$$

First, a *coin* is given, then every execution of the rule eats a *coin*, but only a succesfull execution produces another *coin*: after the first failure, the loop ends. Yet, there are issues: in presence of two or more rules, the *coin* owned by a rule could be used by another one. Furthermore, if object a or c is absent, the process may pause in a spurious state and allow other rules to begin computing before ending execution or releasing back the catalyst c. Even if it is possible to prove that the final result does not change, this breaks the idea of *atomicity* for the application of a rule. Thus, channel *coin* must be private and the priority of every action shall be increased:

$$[\![\ ca \rightarrow cbd\ ]\!]_R = (\nu\ coin)(\overline{coin} \mid !\, coin.(\underline{\tau} + \underline{\underline{c}}.(\underline{\tau}.\overline{\underline{\underline{c}}} + \underline{\underline{a}}.(\overline{\underline{\underline{c}}} \mid \overline{\underline{\underline{b}}} \mid \overline{\underline{\underline{d}}} \mid \overline{coin}))))$$

Maximal parallelism is a heavy constraint on the order of actions: objects may be used only once for every tick of a sort of global clock. From an algorithmic point of view, it may be thought as a global loop of this kind:

```
until (in each tick at least one rule can be applied) do
    until (in current tick some rule can be applied) do
        apply a rule, freezing, for current tick, produced objects
    done
    let clock tick, unfreezing just produced objects
done
```

The outermost loop can be translated with the same scheme used to overcome divergence in the encoding of rules: the only requirement is that rules signal in some way their application performed during each clock tick, in order to allow the loop to continue if at least one rule has been applied at least once:

$$\begin{aligned}[\![\ ca \rightarrow cbd\ ]\!]_R \quad = \quad & (\nu\ coin)(!\ coin.(\underline{\tau}.\underline{\underline{tick}}.\overline{coin} + \underline{\underline{c}}.(\underline{\tau}.(\overline{\underline{\underline{c}}}\ \mid\ \underline{\underline{tick}}.\overline{coin})+ \\ & \underline{\underline{a}}.(\underline{\underline{tick}}(\overline{\underline{\underline{c}}}\ \mid\ \overline{\underline{\underline{b}}}\ \mid\ \overline{\underline{\underline{d}}})\ \mid\ \overline{coin}\ \mid\ \overline{\underline{\underline{worked}}})))\ \mid\ \underline{\underline{tick}}.\overline{coin})\end{aligned}$$

Now, every application of the rule produces a $\overline{\underline{\underline{worked}}}$ process, signaling its application, and the rule waits for its *coin* from the $\overline{\underline{\underline{tick}}}$ of the global clock. When the rule application fails (i.e. when the $\underline{\tau}$ actions are performed), a new process waits for next *coin*. The objects $\overline{\underline{\underline{c}}}, \overline{\underline{\underline{b}}}, \overline{\underline{\underline{d}}}$ produced by the rule are also frozen until next clock $\overline{\underline{\underline{tick}}}$. The global clock can expressed by a low priority loop:

$$!\ clock\!:\!3.\underline{\underline{worked}}.\overline{\underline{\underline{bcast}}}\langle worked\rangle.\tau.\overline{\underline{\underline{bcast}}}\langle tick\rangle.\overline{clock\!:\!3}\ \mid\ \overline{clock\!:\!3}\ \mid\ \overline{\underline{\underline{worked}}}$$

The first instruction, executed with the lowest priority, $clock\!:\!3$, is the guard for the global clock loop, working in the same way of the one previously seen: its *coin* is represented by $\overline{clock\!:\!3}$. As soon as this low priority guard fires, the presence of at least one $\overline{\underline{\underline{worked}}}$ process is checked and the potential duplicates are burned by a subsequent broadcast on the same name. Finally, all the processes waiting for next clock tick (rules waiting for their coin, or frozen objects) are awakened by another broadcast.

Recalling previous considerations about localisation and movement of objects between membranes, the encoding function for rules can finally be formalised:

Definition 4.4 Given an evolution rule $E \in R_{Dom}$, the function

$$[\![\ \ ]\!]_R : R_{Dom} \times I\!N \longrightarrow \mathcal{P}$$

is defined as

$$\begin{aligned}[\![\ E, n\ ]\!]_R \quad \stackrel{def}{=} \quad & (\nu\ coin)(!\ coin.(\underline{\tau}.\underline{\underline{tick}}.\overline{coin} + [\![\ E, n\ ]\!]'_R)\ \mid\ \underline{\underline{tick}}.\overline{coin})\ \mid \\ & \mid\ CLOCK\ \mid\ BCAST\end{aligned}$$

where

$$\begin{array}{rcl} CLOCK & \equiv & !\, clock\!:\!3.\underline{\underline{worked}}.\underline{\underline{\overline{bcast}}}\langle worked\rangle.\tau.\underline{\underline{\overline{bcast}}}\langle tick\rangle.\overline{clock\!:\!3} \;\mid \\ & & \mid\; \overline{clock\!:\!3} \;\mid\; \underline{\underline{\overline{worked}}} \\ \\ BCAST & \equiv & !\, \underline{\underline{bcast}}(x).(\underline{\tau} + \underline{\underline{\overline{x}}}.\underline{\underline{\overline{bcast}}}\langle x\rangle) \end{array}$$

The function

$$\llbracket\;\rrbracket'_R : R_{Dom} \times I\!N \longrightarrow \mathcal{P}$$

is defined as

$$\llbracket\, ca \to cv, n \,\rrbracket'_R \;\stackrel{def}{=}\; \underline{\underline{\overline{c@n}}}.(\underline{\tau}.(\underline{\underline{\overline{c@n}}} \mid \underline{\underline{tick}}.\overline{coin}) + \llbracket\, a \to (c, here)v, n \,\rrbracket'_R)$$

$$\begin{array}{rcl} \llbracket\, a \to v_1 \ldots v_k, n \,\rrbracket'_R & \stackrel{def}{=} & \underline{\underline{\overline{a@n}}}.(\overline{coin} \mid \underline{\underline{\overline{worked}}} \mid \\ & & \qquad\qquad \mid \llbracket\, v_1, n \,\rrbracket''_R \mid \ldots \mid \llbracket\, v_k, n \,\rrbracket''_R) \end{array}$$

where $a \in V \setminus C$, $c \in C$, $v \in (V \times \{here, out, in\})^*$, $v = v_1 \ldots v_n$, V alphabet of the P system, C set of catalysts.
Finally, the function

$$\llbracket\;\rrbracket''_R : (V \times \{here, out, in\}) \times I\!N \longrightarrow \mathcal{P}$$

is defined as

$$\begin{array}{rcl} \llbracket\, (a, here), n \,\rrbracket''_R & \stackrel{def}{=} & \underline{\underline{tick}}.\underline{\underline{\overline{a@n}}} \\ \\ \llbracket\, (a, out), n \,\rrbracket''_R & \stackrel{def}{=} & \underline{out@n}(x).\underline{\underline{tick}}.\underline{\underline{\overline{a@x}}} \\ \\ \llbracket\, (a, in), n \,\rrbracket''_R & \stackrel{def}{=} & \underline{in@n}(x).\underline{\underline{tick}}.\underline{\underline{\overline{a@x}}} \end{array}$$

The definition of $CLOCK$ and $BCAST$ does not depend on any rule, so an immediate optimisation would be insert them in the encoding of the function $\llbracket\;\rrbracket$ in order to avoid useless duplicates, but this affects only the compilation phase.

5 Conclusion

In this paper we defined one possible encoding in $\pi@$ of catalytic P systems. We showed that it is possible to encode in a completely modular way such kind of P systems, using 4 levels of priority and vectors of names of size ≤ 2 for channels.

The encoding preserves all the peculiarities of P systems: nondeterminism, maximal parallelism, divergence/termination. While the first characteristic is an immediate consequence of the intrinsic nondeterminism typical of process calculi, maximal parallelism and termination are reached by means of a quite sophisticated use of priorities. Membrane structure and localisation of objects and rules is expressed by means of polyadic synchronisation.

The encoding of catalytic P systems is only a preliminar step towards more complex and interesting encodings, in particular those pertaining P systems with dynamical membrane structure, in order to investigate the affinities between them and bio-inspired languages like Brane Calculi.

References

[1] M. Carbone, S. Maffeis. On the Expressive Power of Polyadic Synchronisation in pi-calculus. In *Nordic Journal of Computing* 10(2): 70-98, 2003.

[2] L. Cardelli. Brane Calculi - Interactions of Biological Membranes. In *Computational Methods in Systems Biology*, 2004.

[3] R. Cleaveland, G. Lüttgen, V. Natarajan. Priority in Process Algebra. In J.A. Bergstra, A. Ponse, S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001..

[4] R. Milner. The Polyadic π-Calculus: a Tutorial. In F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[5] R. Milner. Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, 1999.

[6] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[7] G. Păun. Introduction to Membrane Computing. First Brainstorming Workshop on Uncertainty in Membrane Computing, Palma de Mallorca, Spain, November 2004, 17-65.

[8] C. Priami, P. Quaglia. Beta binders for biological interactions. In *Computational Methods in Systems Biology*, 2004.

[9] A. Regev, E. Panina, W. Silverman, L. Cardelli, E. Shapiro. BioAmbients: an abstraction for biological compartments. *Theoretical Computer Science*, 2004.

[10] A. Regev, W. Silverman, E. Shapiro. Representing Biomolecular Processes with Computer Process Algebra: π-Calculus programs of Signal Transduction pathways. 2000.

[11] C. Versari. Formal representation of complex biological systems. Master Thesys (in italian), University of Bologna, 2005.