(Ultimately Confluent) Parallel Multiset–Rewriting Systems with Context

Artiom ALHAZOV[1,2], **Dragoş SBURLAN**[3]

[1] Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tárraco 1, 43005 Tarragona, Spain
E-mail: artiome.alhazov@estudiants.urv.es

[2] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Str. Academiei 5, Chişinău, MD 2028, Moldova
E-mail: artiom@math.md

[3] Faculty of Mathematics and Computer Science
Ovidius University of Constanţa
Bd. Mamaia 120, Constanţa, România
E-mail: dsburlan@univ-ovidius.ro

Abstract. The aim of this paper is to study the power of parallel multiset-rewriting systems with permitting context (or P systems with *non-cooperative rules* with *promoters*). The main result obtained is that if we use promoters of weight two, then the system is universal.

Moreover, the construction satisfies a special property we define: it is *ultimately confluent*. This means that if the system allows at least one halting computation, then its final configuration is reachable from any reachable configuration.

1 Introduction

The computational model of membrane computing inspired from the functioning of living cells and formalized through P systems proved to be of a special interest for the scientific community, especially when the weakest forms of cooperation are studied. Activating and prohibiting reactions of various substances (molecules) present in cells is modeled in the P system framework by means of promoters/inhibitors (acting at the level of rules) which enforce/forbid the execution of certain rules. When no mechanism for inhibiting the massive parallel characteristic of a P system exists, a deterministic computation is harder to obtain, especially when non semi-linear languages not belonging to $ET0L$ are studied.

Usually in computer science theory we are interested in solving problems in a predictable time. This is in general a reasonable request if it actually can be done (usually by having a deterministic or a restricted form of nondeterministic computation). However, sometimes it may happen that we are not able to compute the time complexity of the

problem to solve. In this case we would like, at least for a particular problem, to have the result in the limit (you may think of this as a semi-algorithm: the time complexity does not count, and the system gives the correct answer when it halts, or never halts if the correct answer does not exist). This bring us to the scope of the current work.

We assume the reader to be familiar with the fundamentals of membrane computing, see `http://psystems.disco.unimib.it` for the bibliography of the domain. The basic model we study is the transitional P systems with promoted/inhibited non-cooperative rules.

2 Deterministic and Confluent Rewriting

We write $C \Rightarrow C'$ if the system allows a direct transition from an instantaneous description C to an instantaneous description C' (C' is then called a next instantaneous description of C). The relation $\Rightarrow^*$ is a reflexive and transitive closure of $\Rightarrow$. For any rewriting system, we use the word *configuration* to mean any instantaneous description C, reachable from the starting one.

Definition 2.1 *A configuration of a rewriting system is called* halting *if no rules of the system can be applied to it.*

In this paper we will only talk about the rewriting systems, producing the result at halting.

Definition 2.2 *A rewriting system is called* deterministic *if for every accessible non-halting configuration C the next configuration is unique.*

Definition 2.3 *A rewriting system is called* confluent *if either all the computations are non-halting, or there exists a configuration C_h, such that all the computations halt in C_h.*

Notice that if, starting at some configuration C, all the computations halt, then there exists $m \geq 0$, such that all the computations starting from C halt in at most m steps.

We will now introduce a weaker definition of a property of systems, with the computations "unavoidably leading" to the same result, but not necessarily bounded by the number of steps.

Definition 2.4 *A rewriting system is* ultimately confluent *if there exists such a halting configuration C_h, that for any configuration C we have $C \Rightarrow^* C_h$.*

This property implies two facts:

1. the halting configuration is unique (C_h),
2. C_h is reachable from any configuration.

From now on we will only consider rewriting systems producing result at halting. Think of the graph of all reachable configurations (the arc from configuration C to configuration C' means that C can derive C' in one step). The graph may be infinite. The node is called *final* if it has out-degree 0.

The system is *deterministic* if all nodes have out-degree at most one (hence, there is at most one final node). The system is *confluent* if either there are no final nodes, or the

final node is unique, and in that case the graph is finite and does not contain cycles. The *ultimately confluent* system may contain cycles, but either all nodes are non-final, or there is a final node reachable from any configuration. See Figure 1 in Section 5 for an example of such graph.

Example: Consider a system with the initial configuration S and rewriting rules $S \to SA$, $A \to \lambda$, $S \to a$.

Note that the system is not deterministic, and one can choose to apply the first rule an unbounded number of times, but from any configuration it is possible to arrive to the halting one $C_h = a$ by erasing all symbols A and applying the second rule (this is equally true no matter if the system is sequential, concurrent or maximally parallel).

3 Preliminaries

The family of recursively enumerable sets of integer vectors is denoted by $PsRE$.

3.1 Notations: P Systems

A non-coperative rule with promoters of weight at most k is a rule of the form $a \to y|_p$, where $a \in V$, $y \in V^*$, $p \in V^*$, $|p| \leq k$. If $p = \lambda$, we write the rule as $a \to y$.

Let us recall some notations related to the power of P systems. By $r\alpha ctP_m(f)$ we denote the family of languages ($\alpha = L$), vector sets ($\alpha = Ps$) or number sets ($\alpha = N$), which are generated ($c = O$) or accepted (with internal input, $c = I$) by P systems with symbol-objects, restricted to satisfy property r (omitted if none), with at most m membranes with the list of features f.

The features considered in the paper are *ncoo* (with non-cooperative object rewriting rules), and pro_2 (with rule promoters of weight at most 2). The P systems can be restricted to be deterministic ($r = D$). We also introduce the classes of confluent ($r = C$) and ultimately confluent ($r = U$) P systems.

For instance, in this paper we study classes

$$PsOP_1(ncoo, pro_2) \text{ and } UPsIP_1(ncoo, pro_2).$$

3.2 Register Machines

An n-register machine is a construct $M = (n, P, i, h)$ where:

- n is the number of registers;
- P is a set of labeled instructions of the form $(j : op(r), k, l)$ where $op(r)$ is an operation on register r of M; symbols j, k, l belong to the set of labels associated in a one-to-one manner with instructions of P;
- i is the initial label;
- h is the final label.

The instructions allowed by an n-register machine are:

- $(e : inc(r), f, z)$ – add one to the contents of register r and proceed to instruction f or to instruction z ($f = z$ for the deterministic variant);

- $(e : dec(r), f, z)$ – jump to register z if the register r is null; otherwise subtract one from register r and jump to instruction labeled f.
- $(h : halt)$ – finish the computation. This is a unique instruction with label h.

If a register machine $M = (n, P, i, h)$, starting from the instruction labeled i with all registers being empty, stops by halting with value n_j in every register j, $1 \leq j \leq k$ and the contents of registers $k+1, \cdots, n$ being empty, then it generates a vector $(n_1, \cdots, n_k) \in \mathbb{N}^k$. Any recursively enumerable numeric vector set can be generated by a register machine.

A register machine $M = (n, P, i, h)$ accepts a vector $(n_1, \cdots, n_k) \in \mathbb{N}^k$ iff, starting from the instruction labeled i, with register j having value n_j for $1 \leq j \leq k$, and the contents of registers $k+1, \cdots, n$ being empty, the machine stops by the *halt* instruction with all registers being empty. *Deterministic* register machines can accept the family of all recursively enumerable sets of numeric vectors.

Proposition 1 *For any partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ there exists a deterministic $(max\{\alpha, \beta\} + 2)$–register machine M computing f in such a way that, when starting with $(n_1, \cdots, n_\alpha \in \mathbb{N}^\alpha$ in registers 1 to α, M has computed $f(n_1, \cdots n_{alpha}) = (r_1, \cdots r_\beta)$ if it halts in the final label h with registers 1 to β containing r_1 to r_β (and with all other registers being empty); if the final label cannot be reached, $f(n_1, \cdots, n_\alpha)$ remains undefined.*

4 Ultimately Confluent Universality

The following theorem shows the computational universality of P systems with object rewriting context-free rules and promoters. The system we propose simulates the moves of a register machine.

Even if the simulated machine is deterministic, because in our system we do not prevent a way to control the nondeterminism, the method used is to reestablish a previous configuration if the computation went in the "wrong way".

The system may not stop even if we have a computation that it should stop. This is due to the nondeterminism and is the price paid to avoid the use of cooperative (or catalytic) rules which may inhibits the parallelism of the system. However, considering a fair computation, an endless simulation of a finite computation has probability zero. In this way the notion of algorithm (in the framework of total functions) makes sense because, when considering the ultimately confluent variant, the P system will stop with probability 1 if the simulated register machine stops.

Theorem 1 $UPsIP_1(ncoo, pro_2) = PsRE$.

Proof. In order to prove this assertion we will simulate a n–register machine $M = (n, P, i, h)$. The contents of register j will be denoted in our simulation by the multiplicity of the object a_j.

Formally we define the P system $\Pi = (O, [\,_1]\,_1, i, R_1)$, where

$$\begin{aligned} O &= \{a_j \mid 1 \leq j \leq n\} \cup \{h, x, y, k_0, k_1, k_2, k_3, k_4\} \\ &\cup \{e \mid (e : inc(j), f) \in P\} \cup \{e, e_0, e_1, e_2, e_3, e_4 \mid (e : dec(j), f, z) \in P\}, \end{aligned}$$

and R_1 is defined as follows:

For each $(e : inc(j), f) \in P$, R_1 contains the rule $e \to a_j f$.

For each $(e : dec(j), f, z) \in P$, (for clarity, the rules are structured according to the order of application and different cases that may occur)

Step & Case	Rules
1ABCD	$e \to e_0k_0$
2ABCD	$k_0 \to k_1$
2BCD	$e_0 \to e_1\|_{a_j}$
3ABCD	$k_1 \to \lambda$
3A	$e_0 \to z\|_{k_1}$
3BCD	$e_1 \to e_2k_2$ $\quad a_j \to x\|_{e_1}$ $\quad a_j \to y\|_{e_1}$
4BCD	$k_2 \to k_3$
4B	$e_2 \to e_4\|_{yy}$
5B	$x \to a_j\|_{e_4}$ $\quad y \to a_j\|_{e_4}$ $\quad k_3 \to \lambda\|_{e_4}$ $\quad e_4 \to e_1$
5CD	$k_3 \to k_4\|_{e_2}$
5D	$e_2 \to e_3\|_{k_3y}$
6CD	$k_4 \to \lambda$
6C	$x \to a_j\|_{e_2k_4}$ $\quad e_2 \to e_1k_1\|_{k_4}$
6D	$x \to a_j\|_{e_3}$ $\quad e_3 \to f$ $\quad y \to \lambda\|_{e_3}$

For a better understanding, below is the table of configurations structured with respect to the computational steps and cases.

	Case A	Case B	Case C	Case D
t_1	e $e \to e_0k_0$	e, a_j^m $e \to e_0k_0$	e, a_j^m $e \to e_0k_0$	e, a_j^m $e \to e_0k_0$
t_2	e_0, k_0 $k_0 \to k_1$	$e_0, k_0, a_j^m, m \geq 1$ $k_0 \to k_1$ $e_0 \to e_1\|_{a_j}$	$e_0, k_0, a_j^m, m \geq 1$ $k_0 \to k_1$ $e_0 \to e_1\|_{a_j}$	$e_0, k_0, a_j^m, m \geq 1$ $k_0 \to k_1$ $e_0 \to e_1\|_{a_j}$
t_3	e_0, k_1 $e_0 \to z\|_{k_1}$ $k_1 \to \lambda$	$e_1, k_1, a_j^m, m \geq 1$ $k_1 \to \lambda$ $e_1 \to e_2k_2$ $a_j \to x\|_{e_1}$ $a_j \to y\|_{e_1}$	$e_1, k_1, a_j^m, m \geq 1$ $k_1 \to \lambda$ $e_1 \to e_2k_2$ $a_j \to x\|_{e_1}$ $a_j \to y\|_{e_1}$	$e_1, k_1, a_j^m, m \geq 1$ $k_1 \to \lambda$ $e_1 \to e_2k_2$ $a_j \to x\|_{e_1}$ $a_j \to y\|_{e_1}$
t_4	z ready for next instruction	$e_2, k_2, x^r, y^p, r \geq 0, p \geq 2$ $k_2 \to k_3$ $e_2 \to e_4\|_{yy}$	e_2, k_2, x^m $k_2 \to k_3$	e_2, k_2, x^{m-1}, y $k_2 \to k_3$
t_5		$e_4, k_3, x^r, y^p, r \geq 0, p \geq 2$ $x \to a_j\|_{e_4}$ $y \to a_j\|_{e_4}$ $k_3 \to \lambda\|_{e_4}$ $e_4 \to e_1k_1$	e_2, k_3, x^m $k_3 \to k_4\|_{e_2}$	e_2, k_3, x^{m-1}, y $k_3 \to k_4\|_{e_2}$ $e_2 \to e_3\|_{k_3y}$
t_6		$e_1, k_1, a_j^m, m \geq 1$ like t_3	e_2, k_4, x^m $x \to a_j\|_{e_2k_4}$ $e_2 \to e_1k_1\|_{k_4}$ $k_4 \to \lambda$	e_3, k_4, x^{m-1}, y $k_4 \to \lambda$ $x \to a_j\|_{e_3}$ $e_3 \to f$ $y \to \lambda\|_{e_3}$
t_7			$e_1, k_1, a_j^m, m \geq 1$ like t_3	a_j^{m-1}, f ready for next instruction

Before we start explaining the simulation of the subtraction instruction, let us give a glance to the main idea of the algorithm. We start the computation by checking if the

register A is empty or not (i.e., we check if there exists a symbol a). In case is empty we can generate the label of the new instruction to be applied, namely z, therefore we can execute a new instruction of the program. Otherwise (there exists at least one symbol a), in a nondeterministic way, we produce from a^m the multisets x^{m-n} and y^n. Now, in case that $n \neq 1$ then we reestablish the branching configuration by changing back the objects x and y to objects a; therefore the process can start again. This process will last up to the moment when, after splitting the objects a into objects x and y, we will have only one object y. Then, we can continue the computation by deleting the object y and changing back the remaining $m-1$ objects x into objects a. Also, we produce a new object (say f) which represent the label of the new instruction to be executed. In this way, we have correctly simulated the decrement scheme.

We start the computation having inside the region the object i representing the initial label which indicate the first instruction to be executed and the multiset $\{(a_j, n_j) \mid 1 \leq j \leq n\}$ representing the initial contents of registers.

More rigorously, let us see what happens during the computation step by step. Initially it is checked if the register A is empty. This is done by first generating the objects e_0, k_0 when the rule $e \to e_0 k_0$ is applied. In the second step, the object e_0 will be transformed into e_1 iff in the region there exists at least one object a.

In case there is no object a, only the rule $k_0 \to k_1$ is applied. Next, the object k_1 will act as a promoter for the rule $e_0 \to z|_{k_1}$ and, in the same time, will be deleted by the rule $k_1 \to \lambda$.

If in the region there exists at least one object a, the rules $k_0 \to k_1$ and $e_0 \to e_1|_a$ are executed simultaneously in the second step of computation. Now, the rule $e_0 \to z|_{k_1}$ cannot be executed anymore because the object e_0 was already transformed into e_1 in the previous computational step. Therefore, if in the region exists an object e_1, we know for sure that in the region is also at least one object a. As a consequence, in the same step, the rules $a \to x$ or/and $a \to y$ are applied. Due to nondeterminism and because of the maximally parallel mode of functioning of the P systems, all the objects a present in the region will be transformed into objects x and/or y.

In the same time, the object e_1, which descends from the object e, will be transformed into e_2 and k_2 (the object k_2 represents a counter which is useful to reestablish the branching configuration if the computation did not work "well"). Right now the computation can split in three possible directions (the number of rules $a_j \to y|_{e_1}$ is zero, one or more). Let us consider the first case when we will have inside the region the objects x^{m-n} and y^n such that $m - n \geq 0$, $n \geq 2$ and also the objects e_2 and k_2.

Since $n \geq 2$, the rules to be applied in the second step are $k_2 \to k_3$ and $e_2 \to e_4|_{yy}$. Next, the branching configuration is restored by the rules: $x \to a|_{e_4}$, $y \to a|_{e_4}$, $k_3 \to \lambda|_{e_4}$, $e_4 \to e_1$. Recall that promoters can react in the same time with the rules that they promote and also, because of the maximally parallel manner of applying the rules, we successfully restore the branching configuration.

Let us consider the second case, when, in a similar fashion as before, we will have after two computational steps the multiset $e_0, k_0, a^m, m \geq 1$. Then, instead of executing both rules $a \to x|_{e_1}$ and $a \to y|_{e_1}$, only one of them is executed, say $a \to x|_{e_1}$. Therefore, the new configuration is e_2, k_2, x^m and the rule to be applied is $k_2 \to k_3$. Now, since there exists the object e_2 (which in the previous case is transformed in forth step because there exists two objects y) the rule $k_3 \to k_4$ is applied. Once we have the object k_4 we can restore as before the branching configuration because we know for sure that the rules

$a \to x|_{e_1}$ and $a \to y|_{e_1}$ where not applied in a "proper" order.

The third case that may occur represents a successful computation. Recall that the difference between this case and the previous ones occurs after applying the rules $a \to x$ and $a \to y$ when we will have the objects x^{n-1} and y. The computation is the same as in the third case up to the fifth step, when, in addition, the rule $e_2 \to e_4|_{k_3 y}$ is applied. After this, inside the region we will have the objects: x^{m-1}, y, k_4 and e_3. The presence of the object e_3 will drive the computation in the "right" way. The rules that will be applied are as follows: $k_4 \to \lambda$, $x \to a|_{e_3}$, $e_3 \to f$, $y \to \lambda|e_3$, $e_3 \to f$. In this way, starting from the objects e, a^m we have successfully computed f, a^{m-1}. The object f is useful to indicate that the subtraction instruction was successfully applied and to point out the new instruction to be executed.

The simulation of the register machine will continue until the halting instruction is reached (if the simulated machine halts). So, the P system halts on some input if and only if the simulated register machine accepts the corresponding vector. □

Based on the proof above, the following corollary holds.

Corollary 4.1 $PsOP_1(ncoo, pro_2) = PsRE$.

Moreover, the membrane contents in the halting configurations correspond to the value of the partial recursive function computed by the simulated register machine, together with its final label (a "witness" that the computation is finished).

5 Appendix: An Example

Consider a register machine $G = (3, z, h, P)$, with registers a, b, c and instruction set P

$$\begin{array}{rcl}
(z: & dec(a), & p, h), \\
(p: & dec(b), & p', r), \\
(p': & inc(c), & q), \\
(q: & dec(a), & q', f), \\
(q': & dec(a), & p, f), \\
(r: & dec(c), & r', s), \\
(r': & inc(b), & r), \\
(s: & inc(b), & z), \\
(f: & dec(a), & f, f), \\
(h: & halt &).
\end{array}$$

accepting the number set $M = \{n^2 \mid n \geq 0\}$.

The idea of the machine is to repeat subtracting $2 \cdot value(b) + 1$ (i.e., 1, 3, 5, etc.) from register a, while incrementing b. Register c is used as an intermediary for subtraction, and then b is restored from c. Below is a derivation, accepting 4:

$$\begin{array}{c}
(4,0,0,z) \Rightarrow (3,0,0,p) \Rightarrow (3,0,0,r) \Rightarrow (3,0,0,s) \Rightarrow (3,1,0,z) \Rightarrow (2,1,0,p) \Rightarrow \\
(2,0,0,p') \Rightarrow (2,0,1,q) \Rightarrow (1,0,1,q') \Rightarrow (0,0,1,p) \Rightarrow (0,0,1,r) \Rightarrow (0,0,0,r') \Rightarrow \\
(0,1,0,r) \Rightarrow (0,2,0,s) \Rightarrow (0,2,0,z) \Rightarrow (0,2,0,h).
\end{array}$$

Thus, the machine stops if and only if the input number is a perfect square, and in that case the register b will contain the square root of the number, other registers containing zero.

To illustrate both the concept of ultimate confluence and the universality proof, we present Figure 1 representing the graph of configurations of the P system simulating G, reachable from $[_1 aaaaz]_1$. This graph happens to be finite, it contains cycles and a single final node. The final node is reachable from any node.

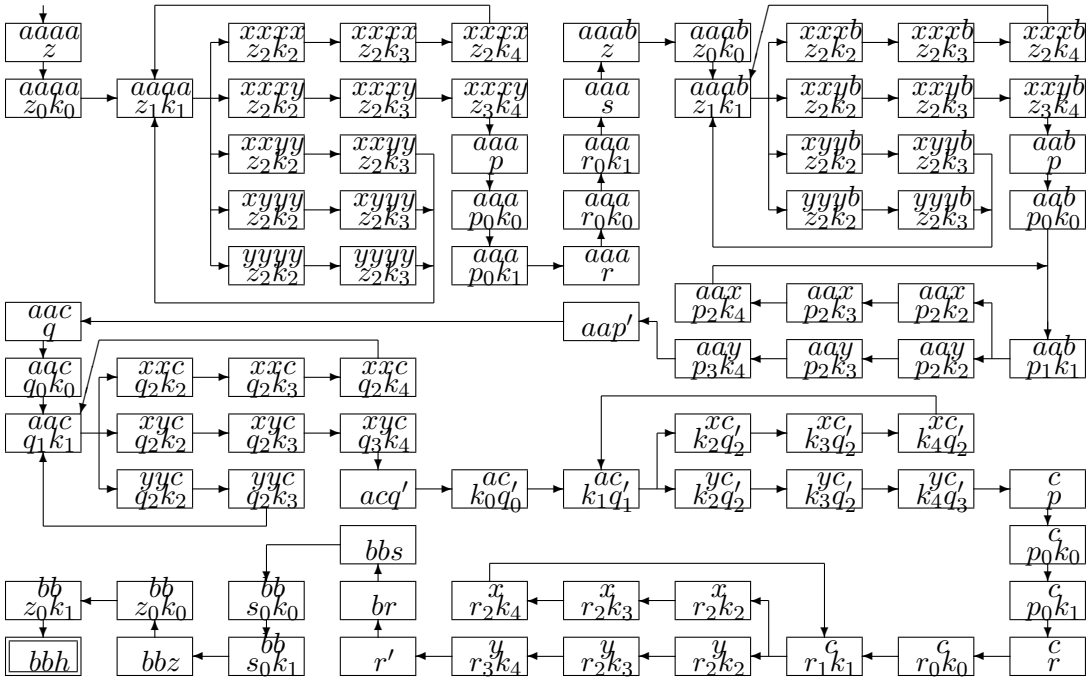

Figure 1: The graph of reachable configurations of a P system simulating register machine G in the ultimately confluent way, recognizing $aaaa$.

Acknowledgements. The first author acknowledges IST-2001-32008 project "MolCoNet" and also the Moldovan Research and Development Association (MRDA) and the U.S. Civilian Research and Development Foundation (CRDF), Award No. MM2-3034.

References

[1] P. Bottoni, C. Martín-Vide, Gh. Păun, G. Rozenberg, Membrane Systems with Promoters/Inhibitors, *Acta Informatica* 38, 10 (2002), 695–720.

[2] Gh. Păun, *Computing with Membranes: An Introduction*. Springer, Berlin, 2002.

[3] G. Rozenberg, A. Salomaa, Eds., *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.