Proceedings of the Workshop on Membrane Computing

WMC 2016

Manchester (UK), 11-15 July 2016

Marian Gheorghe and Savas Konur (Eds.)

School of Electrical Engineering and Computer Science
University of Bradford
Bradford, BD7 1DP, UK

Technical Report – UB-20160819-1

University of Bradford

Contents

Preface

This Workshop on Membrane Computing, at the Conference of Unconventional Computation and Natural Computation (UCNC), 12th July 2016, Manchester, UK, is the second event of this type after the Workshop at UCNC 2015 in Auckland, New Zealand[1]. Following the tradition of the 2015 Workshop the Proceedings are published as technical report.

The Workshop consisted of one invited talk and six contributed presentations (three full papers and three extended abstracts) covering a broad spectrum of topics in Membrane Computing, from computational and complexity theory to formal verification, simulation and applications in robotics. All these papers – see below, but the last extended abstract, are included in this volume.

The invited talk given by Rudolf Freund, "P Systems Working in Set Modes", presented a general overview on basic topics in the theory of Membrane Computing as well as new developments and future research directions in this area.

Radu Nicolescu in "Distributed and Parallel Dynamic Programming Algorithms Modelled on cP Systems" presented an interesting dynamic programming algorithm in a distributed and parallel setting based on P systems enriched with adequate data structure and programming concepts representation. Omar Belingheri, Antonio E. Porreca and Claudio Zandron showed in "P Systems with Hybrid Sets" that P systems with negative multiplicities of objects are less powerful than Turing machines. Artiom Alhazov, Rudolf Freund and Sergiu Ivanov presented in "Extended Spiking Neural P Systems with States" new results regading the newly introduced topic of spiking neural P systems where states are considered.

"Selection Criteria for Statistical Model Checker", by Mehmet E. Bakir and Mike Stannett, presented some early experiments in selecting adequate statistical model checkers for biological systems modelled with P systems. In "Towards Agent-Based Simulation of Kernel P Systems using FLAME and FLAME GPU", Raluca Lefticaru, Luis F. Macías-Ramos, Ionuţ M. Niculescu, Laurenţiu Mierlă presented some of the advatages of implementing kernel P systems simulations in FLAME. Andrei G. Florea and Cătălin Buiu, in " An Efficient Implementation and Integration of a P Colony Simulator for Swarm Robotics Applications" presented an interesting and efficient implementation based on P colonies for swarms of Kilobot robots.

The Workshop organisers would like to thank the Programme Committee members that have contributed with comments and suggestions to the improvement of the contributed papers - Erzsébet Csuhaj-Varjú, Alberto Leporati, Radu Nicolescu, Agustín Riscos-Núñez, Mike Stannett, György Vaszil and Gexiang Zhang.

Marian Gheorghe and Savas Konur

[1] http://ucnc15.wordpress.fos.auckland.ac.nz/workshop-on-membrane-computing-wmc-at-the-conference-on-unconventional-computation-natural-computation/

P Systems Working in Set Modes

Artiom Alhazov[1], Rudolf Freund[2], and Sergey Verlan[3]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md
[2] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Wien, Austria
E-mail: rudi@emcc.at
[3] LACL, Université Paris Est – Créteil Val de Marne
61, av. Général de Gaulle, 94010, Créteil, France
Email: verlan@u-pec.fr

Abstract. In P systems working in the set derivation mode, even in the maximally parallel derivation mode, rules are only applied in at most one copy in each derivation step. We also consider the set mode in the cases of taking those sets of rules with the maximal number of applicable rules or with affecting the maximal number of objects. For many variants of P systems, the computational completeness proofs even literally still hold true for these new set derivation modes. On the other hand, for P systems using target selection for the rules to be chosen together with these set derivation modes we obtain new results.

1 Introduction

In their basic variants, P systems (with symbol objects), usually apply multisets of rules in parallel to the objects in the underlying configuration, i.e., in the maximally parallel derivation mode (abbreviated *max*), a non-extendable multiset of rules is applied to the current configuration. Here we consider the derivation modes, where each rule is only used in at most one copy, i.e., we consider sets of rules to be applied in parallel, for example, in the *set-maximally parallel derivation mode* (abbreviated *smax*) we apply non-extendable *sets* of rules.

Taking sets of rules instead of multisets is a quite natural restriction which has already appeared implicitly in [6] as the variant of the min_1-derivation mode where each rule from its own partition. In an explicit way, the set derivation mode first was investigated in [8] where the derivation mode *smax* was called *flat maximally parallel derivation mode* and where it was shown that in some cases the computational completeness results established for the *max*-mode also hold for the flat maximally parallel derivation mode, i.e., for the *smax*-mode.

In this paper we consider several well-known variants of P systems where the proofs for computational completeness for *max* can be taken over even literally

for $smax$ as well as for the derivation modes max_{rules}, $max_{objects}$ and $smax_{rules}$, $smax_{objects}$, where multisets or sets of rules with the maximal number of rules and multisets or sets of rules affecting the maximal number of objects, respectively, are taken into account. For P systems using target selection for the rules to be chosen these set derivation modes yield even stronger new results. Full proofs of the results mentioned in this paper and a series of additional results can be found in [3].

2 Variants of P Systems

In this section we recall the well-known definitions of several variants of P systems as well as some variants of derivation modes and also introduce the variants of set derivation modes considered in the following.

For all the notions and results not referred to otherwise we refer the reader to the Handbook of Membrane Computing [9].

A (cell-like) P system is a construct

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, f_O, f_I) \text{ where}$$

- O is the alphabet of objects,
- $C \subset O$ is the set of catalysts,
- μ is the membrane structure (with m membranes, labeled by 1 to m),
- $w_1, \ldots, w_m$ are multisets of objects present in the m regions of μ at the beginning of a computation,
- $R_1, \ldots, R_m$ are finite sets of rules, associated with the regions of μ,
- f_O is the label of the membrane region from which the outputs are taken (in the generative case),
- f_I is the label of the membrane region where the inputs are put at the beginning of a computation (in the accepting case).

$f_O = 0/f_I = 0$ indicates that the output/input is taken from the environment. If f_O and f_I indicate the same label, we only write f for both labels.

If a rule $u \to v$ has at least two objects in u, then it is called *cooperative*, otherwise it is called *non-cooperative*. *Catalytic rules* are of the form $ca \to cv$, where $c \in C$ is a special object which never evolves and never passes through a membrane, it just assists object a to evolve to the multiset v.

In *catalytic P systems* we use non-cooperative as well as catalytic rules. In a *purely catalytic P system* we only allow catalytic rules.

2.1 Derivation Modes

In the *maximally parallel derivation mode* (abbreviated by max), in any computation step of Π we choose a multiset of rules from $\mathcal{R}$ (which is defined as the union of the sets $R_1, \ldots, R_m$) in such a way that no further rule can be added to

it so that the obtained multiset would still be applicable to the existing objects in the regions $1, \ldots, m$.

The basic set derivation mode is defined as the derivation mode where in each derivation step at most one copy of each rule may be applied in parallel with the other rules; this variant of a basic derivation mode corresponds to the asynchronous mode with the restriction that only those multisets of rules are applicable which contain at most one copy of each rule, i.e., we consider *sets* of rules:

$$Appl(\Pi, C, set) = \{R \in Appl(\Pi, C, asyn) \mid |R|_r \leq 1 \text{ for each } r \in \mathcal{R}\}$$

In the *set-maximally parallel derivation mode* (this derivation mode is abbreviated by *smax* for short), in any computation step of Π we choose a non-extendable multiset R of rules from $Appl(\Pi, C, set)$; following the notations elaborated in [6], we define the mode *smax* as follows:

$$\begin{aligned} Appl(\Pi, C, smax) = \{R \in Appl(\Pi, C, set) \mid &\text{ there is no } R' \in Appl(\Pi, C, set) \\ &\text{such that } R' \supset R\} \end{aligned}$$

The *smax*-derivation mode corresponds to the min_1-mode with the discrete partitioning of rules (each rule forms its own partition), see [6].

As already introduced for multisets of rules in [4], we now consider the variant where the maximal number of rules is chosen. In the derivation mode $max_{rules}max$ only a maximal multiset of rules is allowed to be applied. But it can also be seen as the variant of the basic mode *max* where we just take a multiset of applicable rules with the maximal number of rules in it, hence, we will also call it the max_{rules} derivation mode. Formally we have:

$$\begin{aligned} Appl(\Pi, C, max_{rules}) = \{&R \in Appl(\Pi, C, asyn) \mid \\ &\text{there is no } R' \in Appl(\Pi, C, asyn) \\ &\text{such that } |R'| > |R|\} \end{aligned}$$

The derivation mode $max_{rules}smax$ is a special variant where only a maximal set of rules is allowed to be applied. But it can also seen as the variant of the basic set mode where we just take a set of applicable rules with the maximal number of rules in it, hence, we will also call it the $smax_{rules}$ derivation mode. Formally we have:

$$\begin{aligned} Appl(\Pi, C, smax_{rules}) = \{&R \in Appl(\Pi, C, set) \mid \\ &\text{there is no } R' \in Appl(\Pi, C, set) \\ &\text{such that } |R'| > |R|\} \end{aligned}$$

We also consider the derivation modes $max_{objects}max$ and $max_{objects}smax$ where from the multisets of rules in $Appl(\Pi, C, max)$ and from the sets of rules in $Appl(\Pi, C, smax)$, respectively, only those are taken which affect the maximal number of objects. As with affecting the maximal number of objects, such

multisets and such sets of rules are non-extendable anyway, we will also use the notations $max_{objects}$ and $smax_{objects}$.

As usual, with all these variants of derivation modes as defined above, we consider halting computations. We may generate or accept or even compute functions or relations. The inputs/outputs may be multisets or strings, defined in the well-known way.

For any derivation mode γ,

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\},$$

the families of number sets ($Y = N$) and Parikh sets ($Y = Ps$) $Y_{\gamma,\delta}(\Pi)$, generated ($\delta = gen$) or accepted ($\delta = acc$) by P systems with at most m membranes and rules of type X, are denoted by $Y_{\gamma,\delta}OP_m(X)$.

3 Computational Completeness Proofs also Working for Set Derivation Modes

In this section we list several variants of P systems where the computational completeness proofs also work for the set derivation modes even being taken literally from the literature.

3.1 P Systems with Cooperative Rules

We first consider *simple P systems with cooperative rules* having only one membrane (the skin membrane), which also serves as input and output membrane, and cooperative rules of the form $u \to v$. Only specifying the relevant parts, we may write $\Pi = (O, w_1, R_1)$ where

- O is the alphabet of objects,
- w_1 is the finite multiset of objects over O present in the skin membrane at the beginning of a computation,
- R_1 is a finite set of cooperative rules.

For a rule $u \to v \in R_1$, $|uv|$ is called its *size*.

Theorem 1. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a simple P system $\Pi = (O, w_1, R_1)$ with cooperative rules of size 3 working in one of the derivation modes from*

$$\{max, max_{rules}, max_{objects}\} \cup \{smax, smax_{rules}, smax_{objects}\}$$

and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R).$$

Proof. Let

$$M = (m, B, l_0, l_h, R)$$

be an arbitrary register machine. We now construct a simple P system with cooperative rules of size 3 simulating M. The number in register r is represented by the corresponding number of symbol objects o_r.

A deterministic ADD-instruction $p : (ADD(r), q)$ is simulated by the rule $p \to o_r q$.

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the two rules $p \to o_r q$ and $p \to o_r s$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated by the following rules:

1. $p \to p'p''$;
2. $p' \to \tilde{p}$, $p''o_r \to \bar{p}$
   (executed in parallel if register is not empty);
3. $\tilde{p}p' \to s$ (if register was empty),
   $\tilde{p}\bar{p} \to q$ (if register was not empty).

In the case of a deterministic register machine, the simulation by the P system is deterministic, too.

We observe that again the construction works for every maximal derivation mode, even if only sets of rules are taken into account. □

3.2 Catalytic and Purely Catalytic P Systems

We now investigate the proofs elaborated for catalytic and purely catalytic P systems working in the *max*-mode for the other (set) maximal derivation modes.

Based on the construction elaborated in [1] we state the following result:

Theorem 2. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a simple catalytic P system*

$$\Pi = (O, C, \mu = [\ \ ]_1, w_1, R_1, f = 1)$$

working in any of the derivation modes γ,

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\},$$

and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 5 \times m + 1.$$

The proof given in [1] can be used for all derivation modes γ, the only exception is that in the set derivation modes in non-successful computations where

more than one trap symbol # has been generated, the trap rule # → # is carried out at most once.

For the purely catalytic case, one additional catalyst c_{m+1} is needed to be used with all the non-cooperative rules. Unfortunately, in this case a slightly more complicated simulation of SUB-instructions is needed, a result established in [11], where for catalytic P systems

$$|R_1| \leq 2 \times ADD^1(R) + 3 \times ADD^2(R) + 6 \times SUB(R) + 5 \times m + 1,$$

and for purely for catalytic P systems

$$|R_1| \leq 2 \times ADD^1(R) + 3 \times ADD^2(R) + 6 \times SUB(R) + 6 \times m + 1,$$

is shown. Yet also this proof literally works for the other (set) derivation modes as well, with the only exception that the trap rule # → # is carried out at most once.

3.3 Computational Completeness of (Purely) Catalytic P Systems with Additional Control Mechanisms

In this subsection we mention results for (purely) catalytic P systems with additional control mechanisms, in that way reaching computational completeness with only one (two) catalyst(s).

P Systems with Label Selection

For all the variants of P systems of type X, we may consider labeling all the rules in the sets $R_1, \ldots, R_m$ in a one-to-one manner by labels from a set H and taking a set W containing subsets of H. In any transition step of a *P system with label selection* Π we first select a set of labels $U \in W$ and then apply a non-empty multiset R of rules such that all the labels of these rules in R are in U in the maximally parallel way. The families of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\},$$

computed by P systems with label selection with at most m membranes and rules of type X is denoted by $Y_{\gamma,\delta}OP_m(X, ls)$.

Theorem 3. *$Y_{\gamma,\delta}OP_1(cat_1, ls) = Y_{\gamma,\delta}OP_1(pcat_2, ls) = YRE$ for any $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and any (set) derivation mode γ,*

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\}.$$

The proof given in [5] for the maximally parallel mode max can be taken over for the other (set) derivation modes word by word; the only difference again is that in set derivation modes, in non-successful computations where more than one trap symbol # has been generated, the trap rule $\# \rightarrow \#$ is only applied once.

Controlled P Systems and Time-Varying P Systems

Another method to control the application of the labeled rules is to use control languages (see [7] and [2]). In a *controlled P system* Π, in addition we use a set H of labels for the rules in Π, and a string language L over 2^H (each subset of H represents an element of the alphabet for L) from a family FL. Every successful computation in Π has to follow a control word $U_1 \dots U_n \in L$: in transition step i, only rules with labels in U_i are allowed to be applied (in the underlying derivation mode, for example, max or $smax$), and after the n-th transition, the computation halts; we may relax this end condition, i.e., we may stop after the i-th transition for any $i \leq n$, and then we speak of *weakly controlled P systems*. If $L = (U_1 \dots U_p)^*$, Π is called a *(weakly) time-varying P system*: in the computation step $pn + i$, $n \geq 0$, rules from the set U_i have to be applied; p is called the *period*. The family of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, computed by (weakly) controlled P systems and (weakly) time-varying P systems with period p, with at most m membranes and rules of type X as well as control languages in FL is denoted by $Y_{\gamma,\delta}OP_m(X, C(FL))$ ($Y_{\gamma,\delta}OP_m(X, wC(FL))$) and $Y_{\gamma,\delta}OP_m(X, TV_p)$ ($Y_{\gamma,\delta}OP_m(X, wTV_p)$), respectively, for $\delta \in \{gen, acc\}$ and

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\}.$$

Theorem 4. *$Y_{\gamma,\delta}OP_1(cat_1, \alpha TV_6) = Y_{\gamma,\delta}OP_1(pcat_2, \alpha TV_6) = YRE$, for any $\alpha \in \{\lambda, w\}$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and*

$$\gamma \in \{sequ, asyn, max, smax\} \cup \{max_{rules}, smax_{rules}, max_{objects}, smax_{objects}\}.$$

The proof given in [5] for the maximally parallel mode max again can be taken over for the other (set) derivation modes word by word, e.g., see [3].

4 Atomic Promoters and Inhibitors

As shown in [10], P systems with non-cooperative rules and atomic inhibitors are not computationally complete when the maximally parallel derivation mode is used. P systems with non-cooperative rules and atomic promoters can at least generate $PsET0L$. On the other hand, already in [8], the computational completeness of P systems with non-cooperative rules and atomic promoters has been shown. In the following we recall our new proof from [3] for the simulation

of a register machine where the overall number of promoters only depends on the number of decrementable registers of the register machine. Moreover, we also recall the proof of a new rather surprising result, establishing computational completeness of P systems with non-cooperative rules and atomic inhibitors, where the number of inhibitors again only depends on the number of decrementable registers of the simulated register machine. Finally, in both cases, if the register machine is deterministic, then the P system is deterministic, too.

4.1 Atomic Promoters

We now recall our new proof from [3] for the computational completeness of P systems with non-cooperative rules and atomic promoters when using any of the set derivation modes $smax, smax_{rules}, smax_{objects}$. The overall number of promoters only is $5m$ where m is the number of decrementable registers of the simulated register machine.

Theorem 5. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with atomic inhibitors*

$$\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$$

working in any of the set derivation modes $smax, smax_{rules}, smax_{objects}$ and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 7 \times m.$$

The number of atomic inhibitors is $5m$. Finally, if the register machine is deterministic, then the P system is deterministic, too.

Proof. The numbers of objects o_r represent the contents of the registers r, $1 \leq r \leq d$; moreover, we denote $B_{SUB} = \{p \mid p : (SUB(r), q, s) \in R\}$.

$$\begin{aligned} O = &\{o_r \mid 1 \leq r \leq d\} \cup \{o'_r, c_r, c'_r, c''_r, c'''_r \mid 1 \leq r \leq m\} \\ &\cup (B \setminus \{l_h\}) \cup \{p', p'', p''' \mid p \in B_{SUB}\} \end{aligned}$$

The symbols from $\{o'_r, c_r, c'_r, c''_r, c'''_r \mid 1 \leq r \leq m\}$ are used as promoters.

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the two rules $p \to qo_r$ and $p \to so_r$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in four steps as follows:

1. $p \to p'c_r$;
2. $p' \to p''c'_r$, $o_r \to o'_r\,|_{c_r}$, $c_r \to \lambda$;
3. $p'' \to p'''c'''_r$, $c'_r \to c''_r\,|_{o'_r}$, $o'_r \to \lambda$;
4. $p''' \to q\,|_{c''_r}$, $p''' \to s\,|_{c'_r}$, $c'_r \to \lambda\,|_{c'''_r}$, $c''_r \to \lambda$, $c'''_r \to \lambda$.

As final rule we could use $l_h \to \lambda$, yet we can omit this rule and replace every appearance of l_h in all rules as described above by λ. □

4.2 Atomic Inhibitors

We now show that even P systems with non-cooperative rules and atomic promoters using the derivation mode $smax, smax_{rules}, smax_{objects}$ can simulate any register machine needing only $2m + 1$ inhibitors where m is the number of decrementable registers of the simulated register machine.

Theorem 6. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with atomic inhibitors*

$$\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$$

a P system with atomic inhibitors $\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$ working in any of the set derivation modes $smax, smax_{rules}, smax_{objects}$ and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 3 \times m + 1.$$

The number of atomic inhibitors is $2m + 1$. Finally, if the register machine is deterministic, then the P system is deterministic, too.

Proof. The numbers of objects o_r represent the contents of the registers r, $1 \leq r \leq d$. The symbols d_r prevent the register symbols o_r, $1 \leq r \leq m$, from evolving.

$$\begin{aligned} O = &\{o_r \mid 1 \leq r \leq d\} \cup \{o'_r \mid 1 \leq r \leq m\} \cup \{d_r \mid 0 \leq r \leq m\} \\ &\cup (B \setminus \{l_h\}) \cup \{p', p'', \tilde{p} \mid p \in B_{SUB}\} \end{aligned}$$

We denote $D = \prod_{i=1}^{m} d_i$ and $D_r = \prod_{i=1, i \neq r}^{m} d_i$.

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the two rules $p \to qo_rD$ and $p \to so_rD$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in four steps as follows:

1. $p \to p'D_r$;
2. $p' \to p''Dd_0$; in parallel, the following rules are used:
   $o_r \to o'_r \,|_{\neg d_r}$, $d_k \to \lambda$, $1 \leq k \leq m$;
3. $p'' \to \tilde{p}D \,|_{\neg o'_r}$; $o'_r \to \lambda$, $d_0 \to \lambda$;
   again, in parallel the rules $d_k \to \lambda$, $1 \leq k \leq m$, are used;
4. $p'' \to qD \,|_{\neg d_0}$, $\tilde{p} \to sD$.

As final rule we could use $l_h \to \lambda$, yet we can omit this rule and replace every appearance of l_h in all rules as described above by λ. □

5 P Systems with Target Selection

In P systems with target selection, all objects on the right-hand side of a rule must have the same target, and in each derivation step, for each region a (multi)set of rules – non-empty if possible – having the same target is chosen. In [3] it was shown that for P systems with target selection in the derivation mode *smax* ***no*** catalyst is needed any more, and with $max_{rules}smax$, we even obtain a deterministic simulation of deterministic register machines.

Theorem 7. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with non-cooperative rules working in the set derivation mode smax and simulating the computations of M.*

When taking the sets of rules with the maximal number of rules which are applicable, the simulation of SUB-instructions can even be carried out in a deterministic way.

Theorem 8. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with non-cooperative rules*

$$\Pi = (O, \mu = [\ [\ ]_2 \dots [\ ]_{2m+1}\ ]_1, w_1, \lambda, \dots, \lambda, R_1 \dots R_{2m+1}, f = 1)$$

working in the derivation mode $smax_{rules}$ and simulating the computations of M such that

$$|R_1| \leq 1 \times ADD^1(R) + 2 \times ADD^2(R) + 4 \times SUB(R) + 10 \times m + 3.$$

Proof. The contents of the registers r, $1 \leq r \leq d$, is represented by the numbers of objects o_r, and for the decrementable registers we also use a copy of the symbol o'_r for each copy of the object o_r. This second copy o'_r is needed during the simulation of SUB-instructions to be able to distinguish between the decrement and the zero test case. For each r, the two objects o_r and o'_r can only be affected by the rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$ sending them into the membrane $r+1$ corresponding to membrane r (and at the same time erasing them; in fact, we could also leave them in the membrane unaffected forever as a garbage). These are already two rules, so any other combination of rules with different targets has to contain at least three rules.

One of the main ideas of the proof construction is that in the skin membrane the label p of an ADD-instruction is represented by the three objects p and e, e', and the label p of any SUB-instruction is represented by the eight objects $p, e, e', e'', d_r, d'_r, \tilde{d}_r, \tilde{d}_r{}'$. Hence, for each $p \in (B \setminus \{l_h\})$ we define $R(p) = pee'$ for $p \in B_{ADD}$ and $R(p) = pee'e''d_r d'_r \tilde{d}_r \tilde{d}_r{}'$ for $p \in B_{SUB}$ as well as $R(l_h) = \lambda$; as initial multiset w_1 in the skin membrane, we take $R(l_0)$.

$$\begin{aligned} O = {} & \{o_r \mid 1 \leq r \leq d\} \cup \{o'_r \mid 1 \leq r \leq m\} \cup (B \setminus \{l_h\}) \\ & \cup \left\{d_r, d'_r, \tilde{d}_r, \tilde{d}_r{}' \mid 1 \leq r \leq m\right\} \cup \{e, e', e''\} \end{aligned}$$

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the rules $p \to R(q)o_r$ and $p \to R(s)o_r$ as well as the rules $e \to \lambda$ and $e' \to \lambda$. This combination of three rules supersedes any combination of rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$, for some $1 \leq r \leq m$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in two steps as follows:

1. In R_1, for the first step we take one of the following tuples of rules:
   $p \to (p, in_{r+1})$, $d_r \to (\lambda, in_{r+1})$, $d'_r \to (\lambda, in_{r+1})$, $\tilde{d_r} \to (\lambda, in_{r+1})$,
   $o_r \to (\lambda, in_{r+1})$, $o'_r \to (\lambda, in_{r+1})$;
   $p \to (p, in_{m+r+1})$, $d_r \to (\lambda, in_{m+r+1})$, $d'_r \to (\lambda, in_{m+r+1})$,
   $\tilde{d_r} \to (\lambda, in_{m+r+1})$, $\tilde{d_r}' \to (\lambda, in_{m+r+1})$;
   the application of the rules $o_r \to (\lambda, in_{r+1})$, $o'_r \to (\lambda, in_{r+1})$ in contrast to the application of the rule $\tilde{d_r}' \to (\lambda, in_{m+r+1})$ determines whether the first or the second tuple of rules has to be chosen. Here it becomes clear why we have to use the two register symbols o_r and o'_r, as we have to guarantee that the target $r+1$ cannot be chosen if none of these symbols is present, as in this case then only four rules could be chosen in contrast to the five rules for the zero test case. On the other hand, if some of these symbols o_r and o'_r are present, then six rules are applicable superseding the five rules which could be used for the zero test case.
2. In the second step, the following three or four rules, again superseding any combination of rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$ for some $1 \leq r \leq m$, are used in the skin membrane:
   $e \to \lambda$, $e' \to \lambda$, $e'' \to \lambda$, and in the decrement case also the rule $\tilde{d_r}' \to \lambda$.
   In the second step, we either find the symbol p in membrane $r+1$, if a symbol o_r together with its copy o'_r has been present for decrementing or in membrane $m+r+1$, if no symbol o_r has been present (zero test case).
   In the decrement case, the following rule is used in R_{r+1}: $p \to (R(q), out)$.
   In the zero test case, the following rule is used in R_{m+r+1}: $p \to (R(s), out)$.

The simulation of the SUB-instructions works deterministically, hence, although the P system itself is not deterministic syntactically, it works in a deterministic way if the underlying register machine is deterministic. □

6 Conclusions

Many of the computational completeness proofs elaborated in the literature for the derivation mode *max* also work for the set derivation mode *smax* and usually even for the other (set) derivation modes max_{rules} and $smax_{rules}$ as well as for $max_{objects}$ and $smax_{objects}$, because many constructions just "break down" maximal parallelism to near sequentiality in order to work for the simulation of register machines. On the other hand, we also have shown that due to this fact some variants of P systems become even stronger with the modes *smax* and $smax_{rules}$. A comprehensive overview of variants of P systems we have already investigated can be found in [3], many more variants wait for future research.

References

1. Alhazov, A., Freund, R.: Small catalytic P systems. In: Dinneen, M.J. (ed.) Proceedings of the Workshop on Membrane Computing 2015 (WMC2015), (Satellite workshop of UCNC2015), August 2015, CDMTCS Research Report Series, vol.

CDMTCS-487, pp. 1–16. Centre for Discrete Mathematics and Theoretical Computer, Science Department of Computer Science University of Auckland, Auckland, New Zealand (2015)
2. Alhazov, A., Freund, R., Heikenwälder, H., Oswald, M., Rogozhin, Yu., Verlan, S.: Sequential P systems with regular control. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, Gy. (eds.) Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7762, pp. 112–127. Springer (2013)
3. Alhazov, A., Freund, R., Verlan, S.: Computational completeness of P systems using maximal variants of the set derivation mode. In: Proceedings 14th Brainstorming Week on Membrane Computing, Sevilla, February 1–5, 2016 (2016)
4. Ciobanu, G., Marcus, S., Păun, Gh.: New strategies of using the rules of a P system in a maximal way. Power and complexity. Romanian Journal of Information Science and Technology 12(2), 21–37 (2009)
5. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013. EPTCS, vol. 128, pp. 47–61 (2013)
6. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. 8th International Workshop, WMC 2007 Thessaloniki, Greece, June 25-28, 2007. Revised Selected and Invited Papers, Lecture Notes in Computer Science, vol. 4860, pp. 271–284. Springer (2007)
7. Krithivasan, K., Păun, Gh., Ramanujan, A.: On controlled P systems. In: Valencia-Cabrera, L., García-Quismondo, M., Macas-Ramos, L., Martínez-del-Amor, M., Păun, Gh., Riscos-Núñez, A. (eds.) Proceedings 11th Brainstorming Week on Membrane Computing, Sevilla, 4–8 February 2013, pp. 137–151. Fenix Editora, Sevilla (2013)
8. Pan, L., Păun, Gh., Song, B.: Flat maximal parallelism in P systems with promoters. Theoretical Computer Science 623, 83–91 (2016)
9. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
10. Sburlan, D.: Further results on P systems with promoters/inhibitors. International Journal of Foundations of Computer Science 17(1), 205–221 (2006)
11. Sosík, P., Langer, M.: Small catalytic P systems simulating register machines. Theoretical Computer Science accepted (2015)

Distributed and Parallel Dynamic Programming Algorithms Modelled on cP Systems

Radu Nicolescu

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
r.nicolescu@auckland.ac.nz

Abstract. We discuss a membrane computing prototype for a simple but typical bottom-up dynamic programming algorithm: finding the longest common subsequence (LCS) of two strings. Conceptually, this problem can be solved by systematically considering all possible subproblems and organising their partial results in a 2D matrix. Large problems can be solved by partitioning this matrix (grid) into blocks, which can be distributed among existing processors. The system evolves by diagonal wavefronts: the blocks are activated by a high-level diagonal wavefront and each active block is swept over by its own diagonal wavefront. We base our work on cP, a slightly revised version of our earlier P systems with complex symbols. We propose a composite prototype of two layers with similar data flows: (i) a message based distributed macro model, and (ii) a shared memory parallel micro model. We discuss the tradeoffs and we conjecture that the same approach can be used to model more complex related algorithms. The asynchronous versions of these prototypes can be efficiently mapped to a distributed Actor system, such as Akka.

Keywords: Dynamic programming, the 13 Berkeley dwarfs, longest common subsequence (LCS), membrane computing, P systems, cP systems, inter-cell parallelism, intra-cell parallelism, Prolog terms and unification, complex symbols, subcells, generic rules, parallel and distributed models, synchronous and asynchronous models, Actor model, Akka.

1 Introduction

We have previously used membrane systems extended with complex symbols (objects) to successfully model a wide variety of applications: image processing and computer vision, graph theory, distributed algorithms, high-level P systems programming, numerical P systems, NP-complete problems. Membrane systems with complex objects include tissue systems as special cases; additionally, they can solve complex problems with fixed sized (and typically small) alphabets and rulesets (independent of the problem size).

For details, please see Nicolescu [10], where a basic image processing task (seeded region growing) is used as a prototype for *structured grid* algorithms,

one of the 13 fundamental classes of parallel patterns, collectively known as the 13 Berkeley dwarfs [3, 2].

In this paper, we propose the cP framework, a slightly revised version of our earlier version of P systems with complex symbols. Using this, we investigate another parallel pattern of the Berkeley collection: *dynamic programming* algorithms. This research is partially based on our earlier modeling exercises related to dynamic programming [11, 5]. In contrast with our earlier papers, here we leverage the power of cP systems to investigate a *composite* design.

We propose a composite prototype consisting of *two layers* with *similar data flows*: (i) a message based *distributed macro* model, and (ii) a shared memory *parallel micro* model. The macro model is a high-level grid having one node for each block. Each macro model node is then mapped to (substituted by) a new instance of the micro model, which is a complex cell, with subcells corresponding to nodes of the original grid (matrix). Essentially, the macro model leverages the *inter-cell* parallelism potential, while the micro model exploits the *intra-cell* potential present in P systems. We discuss various tradeoffs and their effect on the main complexity measures.

The proposed cP model was validated by hand-translation to Akka, a well-known distributed Actor system, with clustering and cloud capabilities. This experiment reinforces our earlier conjectures [9, 8, 10] that (i) membrane systems with complex symbols are adequate for modelling practical parallel and distributed algorithms, succinctly and efficiently (i.e. in "real time"); and (ii) the translation from CP systems to Actors can be largely automatised.

Because of space constraints, for the rest of the paper, we assume some basic familiarity with:

- The basic *longest common subsequence* (LCS) problem and the related *dynamic programming* concepts. Section 2 presents a bird's eye view; for further details see any monograph on algorithms, e.g. [4].
- The *dynamic programming* pattern in *parallel processing*. Section 3 presents a bird's eye view; for further details see the classical Berkeley papers on the 13 parallel dwarfs topic, e.g. [3, 2].
- The *Actor* model in *functional programming*, e.g. as discussed in any Akka tutorial or monograph, e.g. [1].
- The basic definitions used in traditional *tissue-like transition P systems*, including state based rules, weak priority, promoters and inhibitors, e.g. as discussed in the membrane computing handbook [15].
- The membrane extensions collectively known as *complex symbols*, proposed by Nicolescu et al. [13, 12, 14, 10], i.e. complex symbols, generic rules, etc.

However, to ensure some degree of self-containment, our revised extensions are reviewed in Appendix A. The reader is encouraged to check the main changes from our earlier version: simplified definition for complex symbols (subcells); better designed data structures (numbers, associative arrays, lists, trees, and their alternative more readable notations); a standard set of complexity measures.

2 Background: LCS and Dynamic Programming

Given a finite set of finite strings, a common subsequence is subsequence which appears in all given strings. The *longest common subsequence* (LCS) problem finds one of the common subsequences of maximum length. For example, given two strings "acba" and "abcdad", there are two common subsequences of maximal length 3: "aca" and "aba".

The naive algorithm which solves this problem is an archetypal representative of the *bottom-up dynamic programming* family. To simplify the border cases, the two strings are left padded with one extra character. The algorithm also uses a cost matrix C and a pointer matrix P, both of size $m \times n$, where m, n are the lengths of the two padded strings, s and z. The leftmost column and the topmost row of matrix C are initially filled with sentinel 0's. To allow a better visualization, the leftmost column and the topmost row of matrix P are here filled with the chars of s and z (respectively).

This algorithm works in two phases: (i) first, a forward phase which computes the maximum cost; and (ii) next, a backward phase which finds one of the optimal subsequences. Figure 1 shows the forward phase of this algorithm, which systematically fill all cells of the cost and pointer matrices C and P. Cell $C[i, j]$ represents maximum partial cost up to string positions $s[i], z[j]$, while cell $P[i, j]$ points back to one of the optimal paths giving cost $C[i, j]$ (here it enables backtracing of the topmost optimal path).

```
for i = 1 to m-1 do
    for j = 1 to n-1 do
        if  s[i] = z[j] then
            C[i,j] <-  C[i-1,j-1] + 1
            P[i,j] <-  '↖'
        elif C[i-1,j] <  C[i,j-1] then
            C[i,j] <-  C[i,j-1]
            P[i,j] <-  '←'
        else  // C[i-1,j] >=  C[i,j-1] then
            C[i,j] <-  C[i-1,j]
            P[i,j] <-  '↑'
```

Fig. 1: The forward phase of LCS.

Figure 1 shows sample code to evaluate matrices C and P. Note that each cell in matrix C (or P) *depends* on three of its adjacent neighbors, situated in these three directions: N (up), W (left), NW (up+left). These two matrices need not be evaluated row-by-row, as indicated in the sample code of the forward pass; any *dependency* compatible order is correct. For example, one could use a column-by-column evaluation order. Or, a diagonal approach, where a SW-NE diagonal k sweeps over the matrix, from its NW (top-left) corner $C[0, 0]$ to its

SE (bottom-right) corner $C[m-1, n-1]$, ensuring that cells $\{C[i,j] \parallel i+j=k\}$ are all evaluated before cells $\{C[i,j] \parallel i+j=k+1\}$.

Figure 2 shows the matrices C and P evaluated for strings "acba" and "abcdad". The maximum cost is found on $C[4,6] = 3$ and the topmost optimal path can be retraced to "aca".

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0	1	1	2	2	2	2
3	0	1	2	2	2	2	2
4	0	1	2	2	2	3	3

(a) Matrix C

	0	1	2	3	4	5	6
0		a	b	c	d	a	d
1	a	↖	←	←	←	↖	←
2	c	↑	↑	↖	←	←	←
3	b	↑	↖	↑	↑	↑	↑
4	a	↖	↑	↑	↑	↖	←

(b) Matrix P

Fig. 2: Matrices C and P, for input strings "acba" and "abcdad".

3 Background: Parallel Dynamic Programming

As also discussed in the Berkley documentation [3, 2], a dynamic programming algorithm can be parallelised by partitioning the original $m \times n$ grid into $m' \times n'$ blocks ($1 \leq m' \leq m, 1 \leq n' \leq n$) and allocating these blocks to different processing nodes. We exclusively focus on the forward phase of one of the two arrays (C) – including the other array would only add complexity, without any clear benefits for the current discussion.

Here, the final S (bottom) row, E (right) column and SE corner of one block become part of the borderline conditions required by the depending blocks. In physically distributed systems, these sentinel values can be sent via messages. Figure 3 illustrates this approach, where our former $m \times n = 4 \times 6$ sample is partitioned into 6 blocks by $m' = 2$ horizontal bands and $n' = 3$ vertical bands.

The *arrows* suggest the direction of the *data flow*. The thick arrows indicate the actual exchange of messages between blocks. The thin arrows detail the contents of these messages: (i) the full thin arrows are actual part of the messages; and (ii) the dotted thin arrows are only virtual, because they are equivalent to simple compositions of full thin arrows (the diagram is commutative).

The thick arrows between blocks define a partial *dependency* order, showing the possible *activation order* among blocks. In a strict synchronous settings, the active blocks form a SW-NE diagonal, which sweeps over the $m' \times n'$ grid, from the NW corner to the SE corner. Blocks on the same diagonal can run in parallel. In our sample, the synchronous activation diagonal will sweep over the existing block in the following order: (i) block $[0,0]$; (ii) blocks $[0,1], [1,0]$ (in parallel); (iii) blocks $[0,2], [1,1]$ (in parallel); (iv) block $[1,2]$.

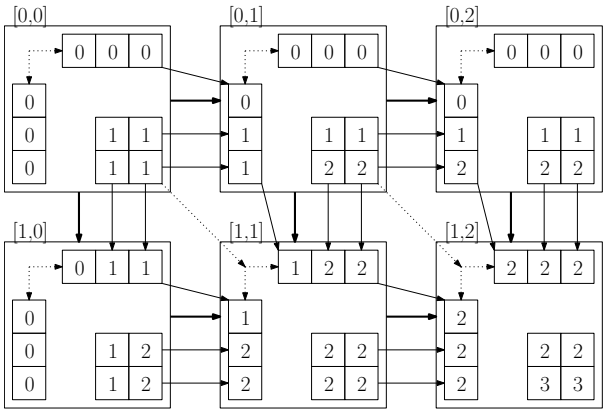


Fig. 3: Original grid partitioned into 6 blocks, by 2×3 bands.

In the asynchronous settings, the activation may get out of the strict diagonal, as permitted by the dependency order. For example, in our sample, the synchronous activation may also include a scenario like the following: (i) block $[0,0]$; (ii) block $[0,1]$; (iii) block $[0,2], [1,0]$ (in parallel); (iii) block $[1,1]$; (iv) block $[1,2]$.

4 cP Macro Model

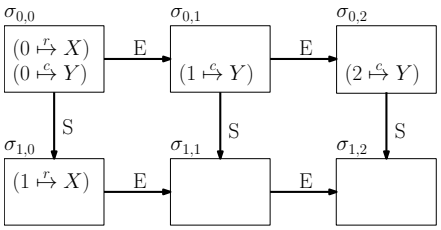
We model the pattern discussed in Section 3 with one cell for each block and arrows indicating the direction of forward messages – note that messages imply a dependency partial order.

We define an *abstract* macro model called $\Sigma_{m',n'}$, which focuses on the messaging pattern and does not perform any real internal computation. The model has $m' \times n'$ cells, $\{\sigma_{i,j} \parallel i \in [0, m'-1], j \in [0, n'-1]\}$. The grid is defined by direct arcs N $\rightarrow$ S and W $\rightarrow$ E.

Initially: (i) each W (left) border cell $\sigma_{i,0}$ is filled with one subcell $r(x(1^i)\ X)$; and (ii) each N (top) border cell $\sigma_{0,j}$ is filled with one subcell $c(y(1^j)\ Y)$ – where X, Y are not yet specified multisets.

Intuitively, terms $x(), y()$ indicate the row index, respectively column index of the current cell and variables X, Y may carry over any additional data required for a concrete instance. For readability, we alias $r(x(1^i)\ X)$, $c(y(1^j)\ Y)$ by the more expressive notations $(i \overset{r}{\mapsto} X)$, $(j \overset{c}{\mapsto} Y)$ (respectively). Figure 4a shows a sample macro model, $\Sigma_{2,3}$, in its initial configuration.

At this abstract level, the evolution rules are simple: each cell is idle until it gets two subcells, one $(I \overset{r}{\mapsto} X')$ and one $(J \overset{c}{\mapsto} Y')$. When this eventuates: (i) the cell becomes active; (ii) creates new subcells X', Y'; (iii) sends one copy of $(I \overset{r}{\mapsto} X')$ to E (right) and one $(J \overset{c}{\mapsto} Y')$ to S (down); and, finally, (iv) becomes



(a) Initial configuration, time=0.

Time	Updates
1	$(0 \overset{r}{\mapsto} X) \Rightarrow \sigma_{0,1}$
1	$(0 \overset{c}{\mapsto} Y) \Rightarrow \sigma_{1,0}$
2	$(0 \overset{r}{\mapsto} X) \Rightarrow \sigma_{0,2}$
2	$(1 \overset{r}{\mapsto} X) \Rightarrow \sigma_{1,1}$
2	$(1 \overset{c}{\mapsto} Y) \Rightarrow \sigma_{1,1}$
3	$(1 \overset{r}{\mapsto} X) \Rightarrow \sigma_{1,2}$
3	$(2 \overset{c}{\mapsto} Y) \Rightarrow \sigma_{1,2}$

(b) Messages received – one possible trace.

Fig. 4: Evolution of $\Sigma_{2,3}$, a 2×3 grid of complex cells.

again idle. Assuming the identity transformation, $X' = X, Y' = Y$, the ruleset can be expressed by the following rules:

$$S_1 \ \rightarrow_{\texttt{min}\otimes\texttt{min}} \ S_2 \ (I \overset{r}{\mapsto} X) \downarrow_E \qquad \| \quad (I \overset{r}{\mapsto} X) \ (J \overset{c}{\mapsto} Y)$$
$$S_1 \ \rightarrow_{\texttt{min}\otimes\texttt{min}} \ S_2 \ (J \overset{c}{\mapsto} Y) \downarrow_S \qquad \| \quad (I \overset{r}{\mapsto} X) \ (J \overset{c}{\mapsto} Y)$$

Figure 4b details one possible evolution of $\Sigma_{2,3}$ (compatible with the induced dependency order). While several evolution orders are possible, $\Sigma_{2,3}$ is confluent and will always reach the same final configuration.

In the *synchronous* setting, this model will activate its cells along the strict diagonal pattern mentioned in Section 3. In the *asynchronous* setting, this model will activate its cells along a "looser" diagonal pattern, where any dependency compliant order will be possible. In both settings, the evolution starts from the NW corner cell $\sigma_{0,0}$ and completes when the SW corner cell $\sigma_{m'-1,n'-1}$ completes, when all the other cells have completed. The following proposition indicates the time and message complexities of a macro model, assuming that we have an unlimited supply of complex cells.

Proposition 1. *The distributed macro system $\Sigma_{m',n'}$ has time complexity $\mathcal{O}(m'+n')$ and message complexity $\mathcal{O}(m'n')$.*

5 cP Micro Model

We now substitute the nodes (blocks) of the abstract macro model $\Sigma_{m',n'}$ by instances of a new model which works "directly" on the original matrix of Section 2. Each $\sigma_{i,j}$ corresponds to an $m_{i,j} \times n_{i,j}$ block $[i,j]$ and is now is mapped to one complex cell $\Theta_{i,j}$. The block dimensions are internally stored as one subcell $\delta(m_{i,j}, n_{i,j})$. Except their dimensions, all these Θ cells are identical.

This cell uses a 2D *associative array* (cf. Appendix A) defined by θ subcells, which have the following general format: $\theta(x(1^i)\ y(1^j)\ s(S)\ z(Z)\ c(C))$,

$0 \leq i \leq m_{i,j}, 0 \leq j \leq n_{i,j}$ – note that there is one extra sentinel row on top (N) and one extra sentinel column on the left (W). The x, y components represent the associative array keys (indexes); the s, z components represent the corresponding chars from the left string, right string (respectively); and the c component represents the corresponding cost in matrix C. For readability, we alias $\theta(x(1^i)\ y(1^j)\ s(s')\ z(z')\ c(c'))$ by the more expressive notation $([i,j] \overset{\theta}{\mapsto} (s', z', c'))$ – the enclosing parentheses may improve the readability, but are not strictly necessary.

Initially, only the topmost and leftmost Θ's contain θ sentinel subcells:
(i) $\forall j \in [0, n]$, $\Theta_{0,j}$ is initialised with $\{[0,k] \overset{\theta}{\mapsto} (s[0], z[k], 0) \parallel k \in [0, m_{0,j}]\}$;
(ii) $\forall i \in [0, m]$, $\Theta_{i,0}$ is initialised with $\{[k,0] \overset{\theta}{\mapsto} (s[k], z[0], 0) \parallel k \in [0, m_{i,0}]\}$.
Other θ subcells will only appear after local Θ computations or from messages sent across neighbouring Θ's.

Each cell Θ remains idle until it receives sentinel subcells $\theta[i,j]$, $i = 0 \vee j = 0$, either from the initial setup or from its N, W neighbours. After becoming active, Θ progressively creates all subcells $\theta[i,j]$, $i > 0 \wedge j > 0$. After *all* θ's have been generated, Θ sends copies of its own S, E border subcells to its S, E neighbours (respectively): its bottom border will become the top row of the S neighbour's associative array, and its rightmost border will become the W leftmost row in the E neighbour's associative array.

Figure 5 shows a sample micro model, $\Theta_{1,1}$, corresponding to block $[1,1]$ of our sample, of size 2×2. Dashed frames enclose sentinel θ's received from its N, W neighbours. Full frames enclose θ's which will be locally created by the evolution rules. Dotted lines represent logical data flow, which will be achieved by local computations (there is no internal messaging). The visual layout of the internal multiset of the θ subcells is intentionally consistent with the conceptual associative array.

The ruleset is a *fixed set* of *six rules*, which work with maximum parallelism (synchronously, as there are no internal messages). First, cell Θ loops in state S_1 until all its θ's are generated. As dictated by the dataflow, each iteration involves the *first three rules* and generates a new SW-NE diagonal, starting from the NW corner. These first three rules represent the bulk of the local computation, closely following the lines of the algorithm of Figure 1. The SE corner $([m_{i,j}, n_{i,j}] \overset{\theta}{\mapsto} (\_, \_, \_))$ is the last one generated; its appearance signals the end of this computing phase.

1. **if** $\nexists$ C[i+1,j+1] **and** s[i+1] = z[j+1] **then** C[i+1,j+1] <− C[i,j] + 1

$$S_1 \ \rightarrow_{\texttt{max}\otimes\texttt{min}} \ S_1' \ ([I1, J1] \overset{\theta}{\mapsto} (S, S, C1)) \quad \neg\, ([I1, J1] \overset{\theta}{\mapsto} (\_, \_, \_))$$
$$\parallel \quad ([I, J] \overset{\theta}{\mapsto} (\_, \_, C)) \quad ([I1, J] \overset{\theta}{\mapsto} (S, \_, \_)) \quad ([I, J1] \overset{\theta}{\mapsto} (\_, S, \_))$$

2. **elif** $\nexists$ C[i+1,j+1] **and** C[i+1,j] >= C[i,j+1] + 1 **then** C[i+1,j+1] <− C[i+1,j]

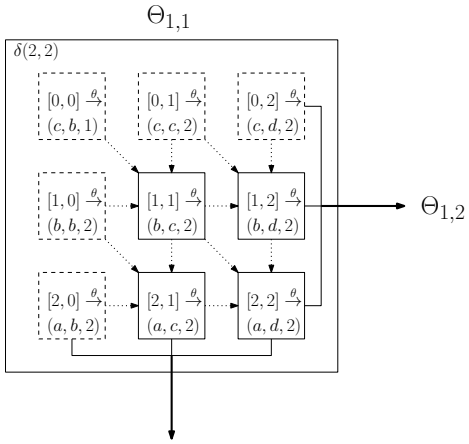


Fig. 5: Final configuration of $\Theta_{1,1}$. Its subcells have been evaluated in the following order: (i) $\theta[1,1]$; (ii) $\theta[1,2]$ and $\theta[2,1]$, in parallel; (iii) $\theta[2,2]$. One message has been sent to its E neighbour $\Theta_{1,2}$. The S message was silently dumped, as there is no S neighbour.

$$S_1 \;\rightarrow_{\texttt{max}\otimes\texttt{min}}\; S_1' \;([I1,J1] \overset{\theta}{\mapsto} (S,Z,C1C')) \quad \neg\,([I1,J1] \overset{\theta}{\mapsto} (\_,\_,\_))$$
$$\| \quad ([I1,J] \overset{\theta}{\mapsto} (S,\_,C1C')) \quad ([I,J1] \overset{\theta}{\mapsto} (\_,Z,C))$$

3. **elif** $\nexists$ C[i+1,j+1] **then** C[i+1,j+1] <− C[i,j+1]

$$S_1 \;\rightarrow_{\texttt{max}\otimes\texttt{min}}\; S_1' \;([I1,J1] \overset{\theta}{\mapsto} (S,Z,C)) \quad \neg\,([I1,J1] \overset{\theta}{\mapsto} (\_,\_,\_))$$
$$\| \quad ([I1,J] \overset{\theta}{\mapsto} (S,\_,\_)) \quad ([I,J1] \overset{\theta}{\mapsto} (\_,Z,C))$$

The above three rules change the state, from S_1 to S_1'; this ensures that only one of these rules is applied. The next and *fourth rule* loops back from state S_1' to S_1:

$$S_1' \;\rightarrow_{\texttt{max}\otimes\texttt{min}}\; S_1$$

The *last two rules* correspond to the ruleset of the macro-model $\Sigma_{m',n'}$ and fire *exactly once*, in the same single step, after the appearance of the SE corner θ. The first rule packs copies of all S border θ's and sends these as one message to Θ's S neighbour. The second rule packs copies of all E border θ's and sends these as one message to Θ's E neighbour. Cell Θ ends in a new idle state, S_2.

$$
\begin{array}{lll}
S_1 & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_2\ ([0,J] \overset{\theta}{\mapsto} (S,Z,C)) \downarrow_S \\
 & & \| \quad ([M,J] \overset{\theta}{\mapsto} (S,Z,C)) \quad ([M,N]) \overset{\theta}{\mapsto} (\_,\_,\_) \quad \delta(M,N) \\
S_1 & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_2\ ([I,0] \overset{\theta}{\mapsto} (S,Z,C)) \downarrow_E \\
 & & \| \quad ([I,N] \overset{\theta}{\mapsto} (S,Z,C)) \quad ([M,N]) \overset{\theta}{\mapsto} (\_,\_,\_) \quad \delta(M,N)
\end{array}
$$

It is straightforward to prove that this model generates its θ subcells following a strict diagonal pattern and its time complexity is linear in its dimensions, assuming that each complex cell has unlimited parallel processing potential.

Proposition 2. *The parallel micro system $\Theta_{i,j}$, has time complexity $\mathcal{O}(m_{i,j} + n_{i,j})$.*

6 Evaluation

Let us assume that all micro-models, $\Theta_{i,j}$, have the same dimension, $m'' \times n''$, i.e. $m_{i,j} = m'', n_{i,j} = n'', \forall (i,j) \in [0, m'-1] \times [0, n'-1]$. This gives the following relations between the lengths of the two strings and the dimensions of our models: $m = m'm'', n = n'n''$. Let $\Upsilon_{m,n,m',n',m'',n''}$ be the final composite model, obtained by substituting the $m' \times n'$ nodes of the distributed abstract model $\Sigma_{m',n'}$ with these equally dimensioned instances of the parallel model $\Theta_{i,j}$, $i \in [0, m'-1], j \in [0, n'-1]$.

Assuming that we have an unlimited supply of complex cells and that each complex cell has unlimited parallel processing potential, the complexity of the final composited model $\Upsilon_{m,n,m',n',m'',n''}$ can be derived from the complexity of its two layers, by combining Proposition 1 and Proposition 2.

Proposition 3. *The distributed and parallel system $\Upsilon_{m,n,m',n',m'',n''}$ has time complexity $\mathcal{O}((m'+n')(m''+n''))$ and message complexity $\mathcal{O}(m'n')$.*

The table in Figure 6 details several interesting scenarios. As generally agreed, overly slow (1,2) or overly chatty (3) scenarios should not be considered (and they may have additional issues). This singles out two scenarios with good theoretical measures:

- Scenario (4) seems the best, but it is not very practical. It requires the availability of huge and powerful complex cells, with very large and fast memories and very massive parallel capabilities. This seems unlikely in the real world (unless we live in a place called Utopia or Shangri-La)...
- Scenario (5) seems the next best one and is probably the most practical. It offers a nice balance between its distributed vs. parallel requirements, so can be used to practically process much larger strings than other alternate scenarios.
  At the first sight, this seems to require $\sqrt{n}\sqrt{n} = n$ distributed nodes. However, this requirement can be reduced to $\sqrt{n}$ nodes. A diagonal sweep over

a $\sqrt{n} \times \sqrt{n}$ grid uses at most only active node per row (or column). Thus, $\sqrt{n}$ active nodes can impersonate all the apparently required n conceptual nodes.

	m'	n'	m''	n''	$\mathcal{O}$ Time complexity	$\mathcal{O}$ Message complexity	Brief notes
0	m'	n'	m''	n''	$(m'+n')(m''+n'')$	$m'n'$	
1	n	1	1	n	n^2	n	Too slow
2	1	n	n	1	n^2	n	Too slow
3	n	n	1	1	n	n^2	Too chatty
4	1	1	n	n	n	1	See discussion
5	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	n	n	See discussion

Fig. 6: Complexity measures for $\Upsilon_{m,n,m',n',m'',n''}$.

7 Conclusions

We discussed a membrane model which solves a typical dynamic programming algorithm in a combined distributed and parallel way. Our proposed model is based on our cP version of the P systems framework. The solution uses a fixed atomic alphabet size, a fixed size ruleset with only *six* rules (!) and has the same time complexity as a much longer hand-written parallel/distributed implementation. Our model can be straightforwardly mapped as a distributed Actor model with cluster and cloud capabilities (Akka.NET).

Together with our previous results, these new results provide additional support for the conjecture that cP systems define expressive, succinct, real-time simulations of hand-written parallel/distributed implementations for practical algorithms and applications.

Like other versions of P systems, our cP systems are formal models which can become directly executable, if properly supported by tools. Further research should address this issue, by reducing the still existing gap between cP systems and the Actor model and by formalising the cP semantics and its translation to a generic Actor model. Ideally, a cP-to-Actors translation should allow the automatic generation of combined message-based distributed macro models with shared-memory parallel micro models and including flexible ways for experimenting with various block granularities.

Acknowledgments. We are deeply indebted to the anonymous reviewers for their valuable comments and suggestions.

References

1. Akka.NET, `http://getakka.net`, `http://getakka.net`, [Online; Revision as of 19:02, 19 April 2016

2. The landscape of parallel computing research: A view from Berkeley (2008), http://view.eecs.berkeley.edu/wiki/Dwarfs, http://view.eecs.berkeley.edu/wiki/Dwarfs, [Online; Revision as of 22:32, 17 November 2008]
3. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (Dec 2006), http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
5. Gimel'farb, G., Nicolescu, R., Ragavan, S.: P systems in stereo matching. In: Real, P., Díaz-Pernil, D., Molina-Abril, H., Berciano, A., Kropatsch, W. (eds.) Computer Analysis of Images and Patterns, Lecture Notes in Computer Science, vol. 6855, pp. 285–292. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-23678-5_33
6. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
7. Nicolescu, R.: Parallel and distributed algorithms in P systems. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Membrane Computing, CMC 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7184, pp. 35–50. Springer Berlin / Heidelberg (2012)
8. Nicolescu, R.: Parallel and distributed seeded region growing with complex objects. In: Sempere, J.M., Zandron, C. (eds.) Proceedings of the 16th International Conference on Membrane Computing (CMC16), 17–21 August, 2015, València (Spain), pp. 271–289. Universitad Politècnica de València, València (Spain) (2015)
9. Nicolescu, R.: Parallel thinning with complex objects and actors. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8961, pp. 330–354. Springer (2015)
10. Nicolescu, R.: Structured grid algorithms modelled with complex objects. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Sixteenth Conference on Membrane Computing (CMC16), Revised Selected Papers, Lecture Notes in Computer Science, vol. 9504, pp. 321–337. Springer (2015)
11. Nicolescu, R., Gimel'farb, G., Morris, J., Gong, R., Delmas, P.: Regularising ill-posed discrete optimisation: Quests with P systems. Fundam. Inf. 131(3-4), 465–483 (2014)
12. Nicolescu, R., Ipate, F., Wu, H.: Programming P systems with complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) 14th Conference on Membrane Computing, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8340, pp. 280–300. Springer (2013)
13. Nicolescu, R., Ipate, F., Wu, H.: Towards high-level P systems programming using complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y. (eds.) 14th International Conference on Membrane Computing, CMC14, Chişinău, Moldova, August 20-23, 2013, Proceedings. pp. 255–276. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, Chişinău (2013)
14. Nicolescu, R., Wu, H.: Complex objects for complex applications. Romanian Journal of Information Science and Technology 17(1), 46–62 (2014)
15. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)

A Appendix
cP Systems : P Systems with Complex Symbols

We present the details of our complex-symbols framework, slightly revised from our earlier papers [9, 10].

A.1 Complex symbols as subcells

Complex symbols play the roles of cellular micro-compartments or substructures, such as organelles, vesicles or cytoophidium assemblies ("snakes"), which are embedded in cells or travel between cells, but without having the full processing power of a complete cell. In our proposal, *complex symbols* represent nested data compartments which have no own processing power: they are acted upon by the rules of their enclosing cells.

Technically, our *complex symbols*, also called *subcells*, are similar to Prolog-like *first-order terms*, recursively built from *multisets* of atoms and variables. *Atoms* are typically denoted by lower case letters (or, occasionally, digits), such as a, b, c, *1*. *Variables* are typically denoted by uppercase letters, such as X, Y, Z. For improved readability, we also consider *anonymous variables*, which are denoted by underscores ("_"). Each underscore occurrence represents a *new* unnamed variable and indicates that something, in which we are not interested, must fill that slot.

Terms are either (i) simple atoms, or (ii) atoms (called *functors*), followed by one or more parenthesized *multisets* (called *arguments*) of other symbols (terms or variables), e.g. $a(b^2X), a(X^2c(Y)), a(b^2)(c(Z))$. Functors that are followed by more than one parenthesized argument are called *curried* (by analogy to functional programming) and, as we see later, are useful to precisely described deep 'micro-surgical" changes which only affect inner nested symbols, without directly touching their enclosing outer symbols. Terms that do *not* contain variables are called *ground*, e.g.:

- Ground terms: a, $a(\lambda)$, $a(b)$, $a(bc)$, $a(b^2c)$, $a(b(c))$, $a(bc(\lambda))$, $a(b(c)d(e))$, $a(b(c)d(e))$, $a(b(c)d(e(\lambda)))$, $a(bc^2d)$; or, a curried form: $a(b^2)(c(d)e^3)$.
- Terms which are not ground: $a(X)$, $a(bX)$, $a(b(X))$, $a(XY)$, $a(X^2)$, $a(XdY)$, $a(Xc())$, $a(b(X)d(e))$, $a(b(c)d(Y))$, $a(b(X)d(e(Y)))$, $a(b(X^2)d(e(Xf^2)))$; or, a curried form: $a(b(X))(d(Y)e^3)$; also, using anonymous variables: $a(b\_)$, $a(X\_)$, $a(b(X)d(e(\_)))$.

Note that we may abbreviate the expression of complex symbols by removing inner λ's as explicit references to the empty multiset, e.g. $a(\lambda) = a()$.

Complex symbols (subcells, terms) can be formally defined by the following grammar:

```
<term> ::= <atom> | <functor> ( '(' <argument> ')' )+
<functor> ::= <atom>
<argument> ::= λ | ( <term-or-var> )+
<term-or-var> ::= <term> | <variable>
```

Natural numbers. Natural numbers can be represented via *bags* containing repeated occurrences of the *same* atom. For example, considering that *1* represents an ad-hoc unary digit, then the following complex symbols can be used to describe the contents of a virtual integer *variable* a: $a() = a(\lambda)$ — the value of a is 0; $a(1^3)$ — the value of a is 3. For concise expressions, we may alias these number representations by their corresponding numbers, e.g. $a() \equiv a(0), b(1^3) \equiv b(3)$. Nicolescu et al. [13, 12, 14] show how arithmetic operations can be efficiently modelled by P systems with complex symbols.

Lists. Using complex symbols, the list $[u, v, w]$ can be represented as $\gamma(u\ \gamma(v\ \gamma(w\ \gamma())))$, where the ad-hoc atom γ represents the list constructor *cons* and $\gamma()$ the empty list. Hiding the less relevant representation choices, we may alias this list by the more expressive notation $\gamma[u, v, w]$.

Trees. Consider the binary tree $z = (a, (b), (c, (d), (e)))$, i.e. z points to a root node which has: (i) the value a; (ii) a left node with value b; and (iii) a right node with value c, left leaf d, and right leaf e. Using complex symbols, tree y can be represented as $z(a\ \phi(b)\ \psi(c\ \phi(d)\ \psi(e)))$, where ad-hoc atoms ϕ, ψ introduce left subtrees, right subtrees (respectively).

Associative arrays. Consider the associative array $\{1 \mapsto a; 1^3 \mapsto c; 1^7 \mapsto g\}$, where the "mapsto" operator, $\mapsto$, indicates key-value mappings. Using complex symbols, this array can be represented as a multiset with three items, $\{\, \mu(\kappa(1)\, \upsilon(a)),\ \mu(\kappa(1^3)\, \upsilon(c)),\ \mu(\kappa(1^7)\, \upsilon(g)) \,\}$, where ad-hoc atoms μ, κ, υ introduce mappings, keys, values (respectively). Hiding the less relevant representation choices, we may alias the items of this multiset by the more expressive notation $\{\, (1 \overset{\mu}{\mapsto} a),\ (1^3 \overset{\mu}{\mapsto} c),\ (1^7 \overset{\mu}{\mapsto} g) \,\} \equiv \{\, 1 \overset{\mu}{\mapsto} a,\ 1^3 \overset{\mu}{\mapsto} c,\ 1^7 \overset{\mu}{\mapsto} g \,\}$. In this context, the "mapsto" operator, $\mapsto$, is considered to have a high associative priority, so the enclosing parentheses are mostly required for increasing the readability (e.g. in text). If we are not interested in the actual mapping value, instead of $(a \overset{\mu}{\mapsto} \_)$, we refer to this term by the succinct abbreviation $x[a]$.

Unification. All terms (ground or not) can be (asymmetrically) *matched* against *ground* terms, using an ad-hoc version of *pattern matching*, more precisely, a *one-way first-order syntactic unification*, where an atom can only match another copy of itself, and a variable can match any bag of ground terms (including the empty bag, λ). This may create a combinatorial *non-determinism*, when a combination of two or more variables are matched against the same bag, in which case an arbitrary matching is chosen. For example:

- Matching $a(b(X)fY) = a(b(cd(e))f^2g)$ deterministically creates a single set of unifiers: $X, Y = cd(e), fg$.
- Matching $a(XY^2) = a(de^2f)$ deterministically creates a single set of unifiers: $X, Y = df, e$.
- Matching $a(XY) = a(df)$ non-deterministically creates one of the following four sets of unifiers: $X, Y = \lambda, df$; $X, Y = df, \lambda$; $X, Y = d, f$; $X, Y = f, d$.

Performance note. If the rules avoid any matching non-determinism, then this proposal should not affect the performance of P simulators running on existing

machines. Assuming that bags are already taken care of, e.g. via hash-tables, our proposed unification probably adds an almost linear factor. Let us recall that, in similar contexts (no occurs check needed), Prolog unification algorithms can run in $O(ng(n))$ steps, where g is the inverse Ackermann function. Our conjecture must be proven though, as the novel presence of multisets may affect the performance.

A.2 Generic rules

Rules use *states* and are applied top-down, in the so-called *weak priority* order. Rules may contain *any* kind of terms, ground and not-ground. In *concrete* models, *cells* can only contain *ground* terms. *Cells* which contain *unground* terms can only be used to define *abstract* models, i.e. high-level patterns which characterise families of similar concrete models.

Pattern matching. Rules are matched against cell contents using the above discussed *pattern matching*, which involves the rule's left-hand side, promoters and inhibitors. Moreover, the matching is *valid* only if, after substituting variables by their values, the rule's right-hand side contains ground terms only (so *no* free variables are injected in the cell or sent to its neighbours), as illustrated by the following sample scenario:

- The cell's *current content* includes the *ground term*:
  $n(a\,\phi(b\,\phi(c)\,\psi(d))\,\psi(e))$
- The following *rewriting rule* is considered:
  $n(X\,\phi(Y\,\phi(Y_1)\,\psi(Y_2))\,\psi(Z)) \;\rightarrow\; v(X)\,n(Y\,\phi(Y_2)\,\psi(Y_1))\,v(Z)$
- Our pattern matching determines the following *unifiers*:
  $X = a$, $Y = b$, $Y_1 = c$, $Y_2 = d$, $Z = e$.
- This is a *valid* matching and, after *substitutions*, the rule's *right-hand* side gives the *new content*:
  $v(a)\;\; n(b\,\phi(d)\,\psi(c))\;\; v(e)$

Generic rules format. We consider rules of the following *generic* format (we call this format generic, because it actually defines templates involving variables):

$$\begin{array}{lll} \textit{current-state}\;\; \textit{symbols}\ldots & \rightarrow_\alpha & \textit{target-state}\;\; (\textit{immediate-symbols})!\ldots \\ & & (\textit{in-symbols})\ldots\;\; (\textit{out-symbols})_\delta\ldots \\ & & \mid \textit{promoters}\ldots\;\; \neg\, \textit{inhibitors}\ldots \end{array}$$

Where:

- All *symbols*, including *states*, *promoters* and *inhibitors*, are *multisets of terms*, possibly containing *variables* (which can be *matched* as previously described).
- Parentheses can be used to clarify the association of symbols, but otherwise have no own meaning.

– Subscript $\alpha \in \{\texttt{min}, \texttt{max}\} \times \{\texttt{min}, \texttt{max}\}$, indicates a combined *instantiation* and *rewriting* mode, as further discussed in the example below.
– *Out-symbols* are sent, at the end of the step, to the cell's structural neighbours. These symbols are enclosed in round parentheses which further indicate their destinations, above abbreviated as δ. The most usual scenarios include:
  • $(a)\downarrow_i$ indicates that a is sent to child i (unicast);
  • $(a)\uparrow_i$ indicates that a is sent to parent i (unicast);
  • $(a)\downarrow_\forall$ indicates that a is sent to all children (broadcast);
  • $(a)\uparrow_\forall$ indicates that a is sent to all parents (broadcast);
  • $(a)\updownarrow_\forall$ indicates that a is sent to all neighbours (broadcast).

  All symbols sent via one *generic rule* to the same destination form one single *message* and they travel together as one single block (even if the generic rule has multiple instantiations).
– Both *immediate-symbols* and *in-symbols* remain in the current cell, but there is a subtle difference:
  • *in-symbols* become available after the end of the current step only, as in traditional P systems (we can imagine that these are sent via an ad-hoc *loopback* channel);
  • *immediate-symbols* become immediately available (i) to the current rule, if it uses the `max` instantiation mode, and (ii) always, to the succeeding rules (in weak priority order).

  Immediate symbols can substantially improve the runtime performance, which could be required for two main reasons: (i) to achieve parity with best traditional algorithms, and (ii) to ensure correctness when proper timing is logically critical. However, they are seldom required and not used in the systems presented in this paper.

Example. To explain our combined instantiation and rewriting mode, let us consider a cell, σ, containing three counter-like complex symbols, $c(1^2)$, $c(1^2)$, $c(1^3)$, and the four possible instantiation⊗rewriting modes of the following "decrementing" rule:

$$(\rho_\alpha)\ S_1\ c(1\,X) \to_\alpha S_2\ c(X), \text{where } \alpha \in \{\texttt{min},\texttt{max}\} \times \{\texttt{min},\texttt{max}\}.$$

1. If $\alpha = \texttt{min}\otimes\texttt{min}$, rule $\rho_{\texttt{min}\otimes\texttt{min}}$ nondeterministically generates and applies (in the `min` mode) *one* of the following two rule instances:

$$(\rho_1')\ \ S_1\ c(1^2) \to_{\texttt{min}} S_2\ c(1) \quad \text{or}$$
$$(\rho_1'')\ \ S_1\ c(1^3) \to_{\texttt{min}} S_2\ c(1^2).$$

   Using (ρ_1'), cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^3)$. Using (ρ_1''), cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.

2. If $\alpha = \texttt{max}\otimes\texttt{min}$, rule $\rho_{\texttt{max}\otimes\texttt{min}}$ first generates and then applies (in the $\texttt{min}$ mode) the following *two* rule instances:

$$(\rho'_2)\ \ S_1\ c(1^2) \rightarrow_{\texttt{min}} S_2\ c(1) \quad \text{and}$$
$$(\rho''_2)\ \ S_1\ c(1^3) \rightarrow_{\texttt{min}} S_2\ c(1^2).$$

   Using (ρ'_2) and (ρ''_2), cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^2)$.

3. If $\alpha = \texttt{min}\otimes\texttt{max}$, rule $\rho_{\texttt{min}\otimes\texttt{max}}$ nondeterministically generates and applies (in the $\texttt{max}$ mode) *one* of the following rule instances:

$$(\rho'_3)\ \ S_1\ c(1^2) \rightarrow_{\texttt{max}} S_2\ c(1) \quad \text{or}$$
$$(\rho''_3)\ \ S_1\ c(1^3) \rightarrow_{\texttt{max}} S_2\ c(1^2).$$

   Using (ρ'_3), cell σ ends with counters $c(1)$, $c(1)$, $c(1^3)$. Using (ρ''_3), cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.

4. If $\alpha = \texttt{max}\otimes\texttt{max}$, rule $\rho_{\texttt{min}\otimes\texttt{max}}$ first generates and then applies (in the $\texttt{max}$ mode) the following *two* rule instances:

$$(\rho'_4)\ \ S_1\ c(1^2) \rightarrow_{\texttt{max}} S_2\ c(1) \quad \text{and}$$
$$(\rho''_4)\ \ S_1\ c(1^3) \rightarrow_{\texttt{max}} S_2\ c(1^2).$$

   Using (ρ'_4) and (ρ''_4), cell σ ends with counters $c(1)$, $c(1)$, $c(1^2)$.

The interpretation of $\texttt{min}\otimes\texttt{min}$, $\texttt{min}\otimes\texttt{max}$ and $\texttt{max}\otimes\texttt{max}$ modes is straightforward. While other interpretations could be considered, the mode $\texttt{max}\otimes\texttt{min}$ indicates that the generic rule is instantiated as *many* times as possible, without *superfluous* instances (i.e. without duplicates or instances which are not applicable) and each one of the instantiated rules is applied *once*, if possible.

If a rule does not contain any non-ground term, then it has only one possible instantiation: itself. Thus, in this case, the instantiation is an *idempotent* transformation, and the modes $\texttt{min}\otimes\texttt{min}$, $\texttt{min}\otimes\texttt{max}$, $\texttt{max}\otimes\texttt{min}$, $\texttt{max}\otimes\texttt{max}$ fall back onto traditional modes $\texttt{min}$, $\texttt{max}$, $\texttt{min}$, $\texttt{max}$, respectively.

Special cases. Simple scenarios involving generic rules are sometimes semantically equivalent to loop-based sets of non-generic rules. For example, consider the rule

$$S_1\ a(x(I)\ y(J)) \ \rightarrow_{\texttt{max}\otimes\texttt{min}}\ S_2\ b(I)\ c(J),$$

where the cell's contents guarantee that I and J only match integers in ranges $[1, n]$ and $[1, m]$, respectively. Under these assumptions, this rule is equivalent to the following set of non-generic rules:

$$S_1\ a_{i,j}\ \rightarrow_{\texttt{min}} S_2\ b_i\ c_j,\ \forall i \in [1, n], j \in [1, m].$$

However, unification is a much more powerful concept, which cannot be generally reduced to simple loops.

Micro-surgery: operations that only affect inner nested symbols. Such operations improve both the crispness and the efficiency of the rules. Consider a

cell that contains symbols $o(abpq), r$ and a naive rule which attempts to change the inner b to a d, if an inner p and a top–level r are also present:

$$S_1\ o(bR)\ \rightarrow_{\texttt{min}\otimes\texttt{min}}\ S_2\ o(dR)\ \mid\ o(p\_)\ r.$$

Unless we change the "standard" application rules, this rule fails, because symbol p is locked as a promoter and cannot be changed at the same time (not even by copy/paste from the left-hand side R to the right-hand side R). We solve this problem without changing the standard application rules, by adding an access path to the inner symbols needed. The *access path* is a slash delimited list of outer symbols, in nesting order, which opens an inner bag for usual rewriting operations; these outer symbols on the path are not themselves touched. For example, this modified rule solves the problem by opening the contents of o for processing:

$$S_1\ o/b\ \rightarrow_{\texttt{min}\otimes\texttt{min}}\ S_2\ o/d\ \mid\ o/p\ r.$$

This extension helps even more when we want to localise the changes to inner symbols of a specific outer symbol. For example, consider a similar operation that needs to be applied on the innermost contents of symbol $o(i,j)(abpq)$, identified by its coordinates i, j.

$$S_1\ o(x(i)\,y(j))/b\ \rightarrow_{\texttt{min}\otimes\texttt{min}}\ S_2\ o(x(i)\,y(j))/d\ \mid\ o(x(i)\,y(j))/p\ r.$$

If all or most symbols involved share the same path, then the path could qualify the whole rule; existing top-level symbols could be qualified by usual path conventions, e.g. in our case, r could be explicitly qualified by either of / or ../:

$$o(x(i)\,y(j))\ ::\ S_1\ b\ \rightarrow_{\texttt{min}\otimes\texttt{min}}\ S_2\ d\ \mid\ p\ ../r.$$

Note that the usual rulesets are just a special case of this extension, when all rules are by default qualified with the root path /.

Note. For all modes, the instantiations are *conceptually* created when rules are tested for applicability and are also *ephemeral*, i.e. they disappear at the end of the step. P system implementations are encouraged to directly apply high-level generic rules, if this is more efficient (it usually is); they may, but need not, start by transforming high-level rules into low-level rules, by way of instantiations.

Benefits. This type of generic rules allow (i) a reasonably fast parsing and processing of subcomponents, and (ii) algorithm descriptions with *fixed size alphabets* and *fixed sized rulesets*, independent of the size of the problem and number of cells in the system (often *impossible* with only atomic symbols).

Synchronous vs asynchronous. In our models, we do not make any *syntactic* difference between the synchronous and asynchronous scenarios; this is strictly a *runtime* assumption [7]. Any model is able to run on both the synchronous and asynchronous runtime "engines", albeit the results may differ.

In the *synchronous* scenario of traditional P systems, all rules in a step take together exactly *one* time unit and then all message exchanges (including loopback messages for in-symbols) are performed at the end of the step, in

zero time (i.e. instantaneously). Alternatively, but logically equivalent, we here consider that rules in a step are performed in *zero* time (i.e. instantaneously) and then all message exchanges are performed in exactly *one* time unit. We prefer the second interpretation, because it allows us to interpret synchronous runs as special cases of asynchronous runs.

In the *asynchronous* scenario, we still consider that rules in a step are performed in *zero* time (i.e. instantaneously), but then, to arrive at its destination, each message may take *any* finite real time in the $(0, 1]$ interval (i.e. travelling times are typically scaled to the travel time of the slowest message). Additionally, unless otherwise specified, we also assume that messages traveling on the same directed arc follow a *FIFO* rule, i.e. no fast message can overtake a slow progressing one. This definition closely emulates the standard definition used for asynchronous distributed algorithms [6]. Clearly, the asynchronous model is highly non-deterministic, but most useful algorithms manage to remain confluent.

In both scenarios, we need to cater for a particularity of P systems, where a cell may remain active after completing its current step and then will automatically start a new step, without necessarily receiving any new message. In contrast, in classical distributed models, nodes may only become active after receiving a new message, so there is no self-activation without messaging. We can solve this issue by (i) assuminging a hidden self-activation message that cells can post themselves at the end of the step and (ii) postulating that such self-addressed messages will arrive not later than any other messages coming from other cells.

Obviously, any algorithm that works correctly in the asynchronous mode will also work correctly in the synchronous mode, but the converse is *not* generally true: extra care may be needed to transform a correct synchronous algorithm into a correct asynchronous one; there are also general control layers, such as *synchronisers*, that can attempt to run a synchronous algorithm on an existing asynchronous runtime, but this does not always work [6].

Complexity measures. We consider a set of basic complexity measures similar to the ones used in the traditional *distributed algorithms* field.

- *Time complexity*: the supremum over all possible running times (which, although not perfect, is the most usual definition for the asynchronous time complexity).
- *Message complexity*: the number of exchanged messages.
- *Atomic complexity*: the number of atoms summed over all exchanged messages (analogous to the traditional bit complexity).

P Systems with Hybrid Sets

Omar Belingheri, Antonio E. Porreca, and Claudio Zandron

Universitá degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
o.belingheri@campus.unimib.it,
{porreca, zandron}@disco.unimib.it

Abstract. In our investigation of the power of a new type of P system which works with objects that can have negative multiplicities, we prove that it is strictly less powerful than a Turing Machine. We get this result by simulating such device using partially blind register machines.

1 Introduction

P systems are a computational model inspired by the structure of a biological cell. Such a model contains several membranes, which can contain several objects; such objects can be transformed and moved from membrane to membrane according to evolution rules. Evolution rules are applied at each step of the computation, changing the configuration of the system until it eventually halts (when no more rules can be applied). The result of a halting computation is the multiset of objects contained in a specified output membrane (or expelled from the system). If the computation does not halt, then it produces no output.

P systems are particularly interesting for their efficiency: trading space for time, we can deal in a polynomial time with problems normally solvable in an exponential time (for further information see [5]). This is possible thanks to the maximal parallel manner in which they operate. A detailed definition of what a P system and its components are can be found in [6].

Several variants of P systems have been investigated through the years. The main differences among the variants usually include the use of new types of rules, or the addition of special abilities to the membranes. However, our variant stands out a little when compared to the others. In fact, we will deal with an aspect regarding the very nature of the system, rather than just focusing on changing specific aspects like its rules. So far the multiplicities representing the objects within a membrane have only been allowed to be nonnegative. In this paper we will let those multiplicities be negative too, expanding the range of their possible values from the set $\mathbb{N}$ to $\mathbb{Z}$ (another attempt to generalize the multiset has been explored in [3]). In doing so we will have to define a new type of P system, called *hybrid P system*, in some ways similar to a catalytic P system (see [6], page 53). After giving some formal definitions necessary to understand hybrid P systems in Section 2, we will go on to investigate their power in Section 3. It will turn out that such systems are less powerful than the Turing Machine, as we will show.

2 Preliminaries

Definition 1. *A P system is a construct*

$$\Pi = (O, \mu, w_1, ..., w_m, R_1, ..., R_m, i_o),$$

where:

1. *O is an alphabet whose elements are called objects;*
2. *μ is the membrane structure with m membranes, labelled with $1, ..., m$;*
3. *w_i, $1 \leq i \leq m$, are strings of the form $a_1^{M(a_1)} a_2^{M(a_2)} ... a_n^{M(a_n)}$ representing the multisets of objects in the regions $1, ..., m$ they are associated with ($M(a_i)$ is the multiplicity of the i-th symbol);*
4. *R_i, $1 \leq i \leq m$ are finite sets of evolution rules. Each R_i associated with region i of μ. Evolution rules are of the form $u \to v$, where u is a string over O and v is a string over $O_{tar} = O \times TAR$, where $TAR = \{here, out\} \cup \{in_j | 1 \leq j \leq m\}$. The symbols here, out, in_j, are called target commands (or target indications) and specify whether an object has to stay in its current membrane (here), go to the outer membrane (out) or must be sent to an inner membrane labelled with j (in_j). Usually we omit the indication "here" for simplicity;*
5. *$i_o \in \{1, ..., m\}$ is the label of an elementary membrane, called output membrane.*

In *hybrid* P systems we allow only standard rules to be used (like the ones we just presented). In such rules only objects with positive multiplicities appear. However we add one condition: objects do not necessarily have to be present for the rules to be applied. This means we can apply a rule regardless of whether its left-hand objects are present with positive multiplicity in a membrane. When doing so, we must subtract from the membrane the amount of objects that were used to apply the rule. For example, if a membrane contains a^2, the rule $a^3 \to bc$ can still be applied, the result of the application being $a^{-1}bc$.

This immediately raises one problem: if the rules can be applied regardless of the presence of an object, then at every step any rule can be applied an infinite amount of times. One possible solution to prevent this from happening, is to add catalysts to the rules. Catalysts are objects present in a finite number in the system, they are used to apply the rules but are not changed by them, and they are not allowed to have negative multiplicity. An example of this kind of rule could be $ak \to ck$, where k is a catalyst.

Before hybrid P systems were conceived, we have always worked with multisets defined as mappings $M : A \longrightarrow \mathbb{N}$, with A an arbitrary set. In order to consider multisets with negative multiplicities, we need to extend that definition to *hybrid sets*, as defined in [1]:

Definition 2. *Given a universe U, any function $f : U \longrightarrow \mathbb{Z}$ is called a hybrid set. As usual, the value $f(x)$ for an element $x \in U$ is said to be the multiplicity of x.*

From now on we will work with hybrid sets instead of multisets. This means it will be normal, and actually very common, for an object to have a negative multiplicity. What do "negative objects" represent in the real world? How can an object be present a negative number of times? In this paper it is not our purpose to answer that question. Our goal is not to model a real cell, but to define and explore new *theoretical* models inspired by cells. However, notice that some physical quantities, such as electrical charge, may indeed have negative integer values and can possibly be modelled by hybrid sets.

Definition 3. *A hybrid P system is defined as*

$$\Pi = (O, K, \mu, w_1, ..., w_m, R_1, ..., R_m, i_o)$$

where all components are defined as for standard P systems (Definition 1), except that all multisets are replaced by hybrid sets. We also have an alphabet of objects $K \subseteq O$ used as catalysts; we require $M(k) \geq 0$ for all $k \in K$ and all hybrid sets M. Furthermore, the evolution rules are of the form $uk \rightarrow vk_t$, where $u \in O^\star$, $k \in P$ and v is a string over O_{tar}, where $O_{tar} = (O - K) \times TAR$, for $TAR = \{here, out\} \cup \{in_j | 1 \leq j \leq m\}$, and $t \in TAR$.

A *configuration* of a system is the m-tuple of hybrid sets of objects present in the system in its m regions. The *initial configuration* of a system is $(w_1, ..., w_m)$.

As time passes, the configuration of a system changes thanks to the application of the evolution rules. A global clock is assumed to exist. Its function is to mark the time for each membrane within the system, dividing the computation in steps. At each step, the evolution rules are applied in a *non-deterministic* and *maximally parallel* manner. This means that the rules to be applied are chosen in a non-deterministic way within every membrane, and every object that can use a rule to evolve must do so.

A rule $R_i : uk \rightarrow vk_t$ can be applied if k is present in membrane i. If it is, the hybrid set u is removed from the membrane and the objects in v are introduced, according to their multiplicity and target commands, and k is (possibly) moved according to t. If v contains the pair (a, in_j) but j is not a membrane immediately inside i, then that rule cannot be applied. Whenever an object is sent out of the skin into the environment, it cannot come back.

Given two configurations $C_1 = (w'_1, ..., w'_m)$ and $C_2 = (w''_1, ..., w''_m)$ of the same system Π, we say that we have a transition from C_1 to C_2 if we can pass from C_1 to C_2 using the evolution rules $R_1, ..., R_m$ in the regions of the system (i.e. we simultaneously transform the hybrid sets in C_1 to obtain C_2). This can be written as $C_1 \Longrightarrow C_2$.

A *computation* is a sequence of transitions between configurations of a system. It is considered successful if and only if the system halts (i.e. no further rules can be applied). In hybrid P systems, this can only happen when all catalysts have been moved to regions where no rule involves them. Once it halts, the result is given by the multiplicities of objects in the output membrane i_o of the system. A computation that does not halt produces no output. P systems can be seen as generators of numbers: we then denote by $Z(\Pi)$ the set of numbers that

can be computed by the system, that is, all the possible vectors of multiplicities of objects in the output membrane when the computation stops.

3 Negative multiplicities weaken the power of a system

We now prove that the use of hybrid sets, where objects are allowed to have negative multiplicities, weakens P systems. Informally, one can expect such a result because of the possibility to transform rules like $uk \to vk_t$ with target $t \in \{here, out, in_j\}$ into the form $k \to u^{-1}vk_t$, that is, we are losing the power of the cooperation by moving u on the other side of the rule with opposite multiplicities.

3.1 Partially blind register machines

We need to formally introduce a particular type of register machine (as in [2]) that is said to be *partially blind*.

We start from a register machine $R = (m, B, l_0, l_h, P)$, where $m \geq 1$ is the number of counters, B is the finite set of instruction labels, l_0 is the initial label, l_h is the halting label, and P is the program, a finite set of instructions from B.

There are three types of instructions:

- $l_1 : (ADD(r), l_2, l_3)$, $1 \leq r \leq m$;
- $l_1 : (SUB(r), l_2, abort)$, $1 \leq r \leq m$ (if r was not empty, then go to l_2, otherwise the machine aborts without producing any result);
- $l_h : HALT$ (which halts the machine).

The main feature of this register machine is that the subtracting instruction does not check if the register is empty. When the RM executes a subtraction from an empty register, it aborts the computation. Technically, even though there does not seem to be a test for zero, this test is implicit: at the end of a successful computation we require some specified registers to be empty; all computations where this does not happen are discarded as aborted computations.

It is known that partially blind register machines are strictly less powerful than Turing machines [4].

3.2 Simulation using partially blind register machines

The language of hybrid sets (resp., the set of vectors of integers) generated by a system Π is denoted by $L(\Pi)$ and it consists of all hybrid sets (resp., vectors of multiplicities of hybrid sets) placed in the output membrane at the end of halting computations. We will denote by NHP and $NPBRG$ the sets generated by a hybrid P system and a partially blind register machine, respectively. We now prove that $NHP \subseteq NPBRG$.

Theorem 1. *Let L be a vector set of numbers generated by a hybrid P system $\Pi = (O, K, \mu, w_1, ..., w_m, R_1, ..., R_m, i_o)$. Then, there exist a $PBRM$ that can generate L.*

Proof. We can build a partially blind register machine

$$R = (2m \cdot |O - K| + 2 \cdot |O - K|,\ B,\ 1,\ 4,\ P)$$

generating L as follows:

1. For each $a \in O - K$ there are pairs of registers labelled a_i^+ and a_i^- such that the value $a_i^+ - a_i^-$ is the multiplicity of object a in membrane i, with $1 \leq i \leq m$ (a total of $2m \cdot |O - K|$ registers). Note that we do not balance out positive and negative occurrences of objects until the computation is over (plus, we do this only in the output membrane); so it will be normal to have the registers associated with the positive and negative multiplicities of an object both nonnegative at the same time. This has no effect on the computation: in fact, the multiplicities of non-catalysts are basically irrelevant until the system stops and they become part of the result;
2. For each $a \in O - K$ we add two output registers labelled $output_{a^+}$ and $output_{a^-}$. The register $output_{a^+}$ (resp., $output_{a^-}$) will contain the absolute value of the final multiplicity of object a in the output membrane, if such multiplicity is positive (resp., negative), and zero otherwise.

The simulation works like this: in the first part, the registers from (1) are used to keep track of how many non-catalyst objects are being generated (resp., deleted); this is performed by increasing the registers a_i^+ (resp., increasing the registers a_i^-) without ever decreasing those registers.

The number of catalysts never changes during the computation, but they can only be moved around the membrane structure in a finite number of distinct configurations. Hence, we can keep track of the position of the catalysts as part of the label of the current instruction of the register machine. In particular, some instruction labels correspond to configurations of catalysts where no rule is applicable and thus Π halts its computation.

Hence, each configuration of catalysts enabling one or more rules of Π corresponds to a set of instructions of R updating the counters of (1) according to those rules. In general, several possible maximally parallel choices of rules competing for the catalysts are possible; however, since these are finite in number, one of them can be nondeterministically chosen by jumping to a corresponding instruction label. At the end of this update, the register machine jumps to the first label corresponding to the new configuration of catalysts moved by the rules they enabled (this might be the same as the current one if no catalyst has been moved at this time step).

Then, when the P system cannot apply any more rules, because all catalysts have been moved to a region where they have no applicable rules (this can be detected by the label of the current instruction of R) we enter a second phase of the simulation. The purpose of this phase is that of levelling the values contained in the pairs of registers a_o^+ and a_o^- (the ones associated with the output membrane): that is, we calculate $|a_o^+| - |a_o^-|$ and leave the result in a_o^+ (if the multiplicity is nonnegative) or in a_o^- (otherwise). This process is performed via a nondeterministically guessed number of subtractions, since we cannot perform a

zero test (if the number of subtractions leads to decrementing an empty register, we simply obtain an aborted computation); the correct result of the subtraction will be checked by exploiting the halting condition of the register machine.

In the third phase of the simulation we copy the results in $output_{a^+}$ or $output_{a^-}$ while deleting the contents of a^+ or a^- respectively. This is also performed by repeated subtractions without zero testing (i.e., a nondeterministic number of times). Finally, the machine halts.

We require all the registers a_o^+, a_o^- $(\forall a \in O - K)$ to be zero when the machine halts. This way we can make sure that two things have happened: (1) the subtraction $|a_o^+| - |a_o^-|$ has been performed correctly, and (2) the multiplicities of the output objects have been copied correctly to the registers $output_{a^+}$ or $output_{a^-}$.

To generate the instructions in P, we proceed as follows. First of all, we take all the rules of the form $uk \to vk_t$ and we transform them into the form $k \to u^{-1}vk_t$. We can then think of grouping the rules depending on the configuration of catalyst that enable them and depending on simultaneous applicability, and generate all the possible combinations of rules that can be applied in one step. Once we have generated all the combinations, we can translate them into instructions for the machine, adding and subtracting the elements from their respective registers.

From the previous discussion, it is easy to see that the P systems can be simulated by the $PBRM$, generating the same vector set of numbers. □

We show a simple example to clarify the process just described in the proof. The $PBRM$ starts in a configuration that reflects the initial state of the P system, which means that all of its registers have been initialized with the multiplicities of the non-catalyst objects they represent, and that the initial instruction label encodes the initial configuration of the catalysts. At each step of the computation, the instructions from the first phase are used until the machine enters the second phase, which calculates the final multiplicity of each non-catalyst object (a, b, c). Finally, the machine "copies" these values into the output registers using the instructions of the third phase and halts (the copy is attained by means of subtracting and adding 1 to the registers).

Example 1. Let us assume to have only one membrane with label 1 and the following rules (already in context-free notation):

$$k \to a^{-1}b^2k \qquad k \to k_{out} \qquad \ell \to b^{-1}c\ell \qquad \ell \to \ell_{out}$$

and that this membrane only contains the catalysts k and ℓ with multiplicity 1 in the initial configuration. Thus, there exist only 4 possible configurations of catalysts, depending on which have been already sent out. When both have been sent out, the computation of the P system halts. All catalyst configurations (except the halting one) involve one or two conflicting rules: each catalyst can either remain where it is and rewrite some objects, or be sent out.

For instance, the block of instructions corresponding to the catalyst configuration where both catalysts are still inside the membrane and the rules

$k \to a^{-1}b^2k$ and $\ell \to b^{-1}c\ell$ are applied is

$$\begin{aligned}
l_1 &: (ADD(a_1^-), l_{11}, l_{11}) \\
l_{11} &: (ADD(b_1^+), l_{12}, l_{12}) \\
l_{12} &: (ADD(b_1^+), l_{13}, l_{13}) \\
l_{13} &: (ADD(b_1^-), l_{14}, l_{14}) \\
l_{14} &: (ADD(c_1^+), l_0, l_0)
\end{aligned}$$

and the computation proceeds with label l_0, where another maximally parallel choice of rules enabled by the same catalyst configuration is nondeterministically made. If the catalysts had been moved, the computation would instead have jumped to the first label corresponding to the new configuration of catalysts.

Now suppose that the P system reaches a configuration where both catalysts have been sent out (thus no more rule is applicable), and suppose that this catalyst configuration corresponds to label l_4. Now the registers corresponding to the output region (membrane 1 in this case) are levelled by repeatedly applying the following instructions for each object type x:

$$\begin{aligned}
l_x &: (SUB(x_1^+), l_{x2}, abort) \\
l_{x2} &: (SUB(x_1^-), l_{x3}, abort)
\end{aligned}$$

where the instructions at label l_{x3} nondeterministically guess whether one of the two registers x_1^+ and x_1^- have reached zero (and thus the difference has been computed correctly) and, if so, the computation proceeds with the following object type instead of repeating the previous two instructions again.

When all pairs of object registers have been levelled, the register machine guesses, for each object type x, whether the final multiplicity is nonnegative (resp., negative) and copies the value of register x_1^+ (resp., x_1^-) into the output register $output_{x^+}$ (resp., $output_{x^-}$). This is, once again, performed by repeated decrement and increment instructions until nondeterministically guessing that the source register is empty.

This way the machine reaches the halt instruction and succeeds if and only if the P system has stopped and its simulation has been performed correctly; a simulation error is detected either when decrementing a register below zero, or when some registers x_1^+ or x_1^- are nonempty in the final configuration of the register machine.

It is well known that $PBRG \subset RE$, where RE denotes the recursively enumerable languages. Hence, an immediate consequence of Theorem 1 is that hybrid P systems are strictly less powerful than Turing machines.

Corollary 1. *$NHP \subset RE$.*

4 Conclusion

In this paper we introduced a new type of P system, which because of its different nature required new definitions. However, there are other variants that can be similarly conceived.

The first variant is a P system where we allow rules to use objects with negative multiplicities; that is, we allow rules like $u^{-1} \to bc^{-1}$. We do not, however, allow the application of any rules when the left-hand objects are not present (like we do in first-type systems). We conjecture that since their rules cannot be applied without the required left-hand objects, their power is not weakened by the use of hybrid sets. The main problem with systems introduced in this paper is that we can use rules at any time, and that makes us lose control over when to stop using a rule.

The second variant is a combination of the other two: in these systems, rules can deal with objects with negative multiplicities and can also be applied at any time. To limit the use of a certain rule, catalysts need to be introduced (or alternatively, one may think of a different way to prevent the computation from never ending). It is unclear how powerful this device would be, and we leave all possible paths open.

References

1. Carette, J., Sexton, A.P., Sorge, V., Watt, S.M.: Symbolic domain decomposition. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) Intelligent Computer Mathematics, 10th International Conference, AISC 2010. Lecture Notes in Computer Science, vol. 6167, pp. 172–188. Springer (2010)
2. Freund, R., Ibarra, O.H., Păun, Gh., Yen, H.C.: Matrix languages, register machines, vector addition systems. In: Gutiérrez-Naranjo, M.A., Riscos-Núñez, A., Romero-Campero, F.J., Sburlan, D. (eds.) Third Brainstorming Week on Membrane Computing. Fénix Editora (2005)
3. Freund, R., Ivanov, S., Verlan, S.: P systems with generalized multisets over totally ordered Abelian groups. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing, 16th International Conference, CMC 2015. Lecture Notes in Computer Science, vol. 9504, pp. 117–136. Springer (2015)
4. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science 7, 311–324 (1978)
5. Păun, Gh.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics 6(1), 75–90 (2001)
6. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002)

Extended Spiking Neural P Systems with States

Artiom Alhazov[1], Rudolf Freund[2], and Sergiu Ivanov[3]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Str. Academiei 5, Chişinău, MD 2028, Moldova
E-mail: artiom@math.md
[2] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: rudi@emcc.at
[3] Université Paris Est, France
E-mail: sergiu.ivanov@u-pec.fr

Abstract. We consider (extended) spiking neural P systems with states, where the applicability of rules in a neuron not only depends on the presence of sufficiently many spikes (yet in contrast to the standard definition, no regular checking sets are used), but also on the current state of the neuron. Moreover, a spiking rule not only sends spikes, but also state information to the connected neurons. We prove that this variant of the original model of extended spiking neural P systems can simulate register machines with only two states, even in the basic non-extended variant.

1 Introduction

In the area of P systems, the model of *spiking neural P systems* was introduced in [6]. Whereas the basic model of membrane systems, see [10], reflects hierarchical membrane structures, in spiking neural P systems the cells are arranged in a tissue-like manner, with the contents of a cell (neuron) consisting of a number of so-called *spikes*, i.e., of a multiset over a single object. The rules assigned to a neuron allow us to send information to other neurons in the form of electrical impulses (also called spikes) which are summed up at the target neuron; the application of the rules depends on the contents of the neuron and in the general case is described by regular sets. As inspired from biology, the neuron sending out spikes may be "closed" for a specific time period corresponding to the refraction period of a neuron; during this refraction period, the neuron is closed for new input and cannot get excited ("fire") for spiking again.

The length of the axon may also cause a time delay before a spike arrives at the target. Moreover, the spikes coming along different axons may cause effects of different magnitude. All these biologically motivated features were included in the model of extended spiking neural P systems considered in [3], the most important theoretical feature being that neurons can send spikes along the axons with different magnitudes at different moments of time.

In this paper, we consider a variant of the model of extended spiking neural P systems which not only uses spikes to be sent to other neurons when some neuron spikes, but also allows for sending some additional information called "state" along the axons. All these state informations arriving in a neuron then determine the next state of the neuron. On the other hand, we do not use the regular checking sets for the current number of spikes in the neuron any more, which decreases the amount of information a spiking rule may use. Hence, the spiking rules now depend on the current states of the neurons and the availability of sufficiently many spikes.

This variant of extended spiking neural P systems with states has been inspired by the variant of spiking neural P systems with polarizations, see [14], where the states are called polarizations, and the underlying model of extended spiking neural P systems was the basic one with a fixed connection structure, only extended by allowing more than one spike to be sent along the axons. There it was shown that computational completeness (i.e., simulation of register machines) can be obtained with only three polarizations. In this paper we now show that computational completeness can already be obtained with only two states, i.e., with two polarizations, even for the basic non-extended variant as considered in [14], which solves an open problem raised at the Brainstorming Week on Membrane Computing in Sevilla at the beginning of February 2016.

The rest of the paper is organized as follows: In the next section, we recall some preliminary notions and definitions from formal language theory, especially the definition and some well-known results for register machines. In Section 3, based on the model of extended spiking neural P systems as considered in [3] we define the model of *spiking neural P systems with states* we use in this paper. In Section 4, we prove our main result and show that spiking neural P systems with only two states (0 and 1) can simulate register machines; the complexity of the construction depends on the features we require the spiking neural P systems to have, but the result even holds true for the basic non-extended variant of spiking neural P systems with states, where the connection structure between the neurons is fixed and does not depend on the spiking rules applied in the neurons, which makes our result comparable we that one obtained in [14] where three states (there called polarizations) were needed. Moreover, we only use a very simple global state composition function computing the new state of a neuron from the state information having arrived from the input neurons in the simplest way by going to the "activated state 1" if and only if at least one such activating signal 1 has come in the previous step. In Section 5, we show how small universal spiking neural P systems with states can be constructed based on the results obtained in this paper. A short summary of the results we obtained concludes the paper.

2 Preliminaries

In this section we recall the basic elements of formal language theory and especially the definitions and some well-known results for register machines; we

also refer to the corresponding section from [3] and [2]. For the basic elements of formal language theory needed in the following, we refer to any monograph in this area, in particular, to [12]. We just list a few notions and notations:

V^* is the free monoid generated by the alphabet V under the operation of concatenation and the empty string, denoted by λ, as unit element; for any $w \in V^*$, $|w|$ denotes the number of symbols in w (the *length* of w). $\mathbb{N}_+$ denotes the set of positive integers (natural numbers), $\mathbb{N}$ is the set of non-negative integers, i.e., $\mathbb{N} = \mathbb{N}_+ \cup \{0\}$.

2.1 Register Machines

The proofs of the results establishing computational completeness in the area of P systems are often based on the simulation of register machines; we refer to [8] for original definitions, and to [5] for the definitions we use in this paper:

An *n-register machine* is a tuple $M = (n, B, l_0, l_h, P)$, where n is the number of registers, B is a set of labels, $l_0 \in B$ is the initial label, $l_h \in B$ is the final label, and P is the set of instructions bijectively labeled by elements of B. The instructions of M can be of the following forms:

- $p : (\texttt{ADD}(r), q, s)$, with $p \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq n$.
  Increases the value of register r by one, followed by a non-deterministic jump to instruction q or s. This instruction is usually called *increment*.
- $p : (\texttt{SUB}(r), q, s)$, with $p \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq n$.
  If the value of register r is zero then jump to instruction s; otherwise, the value of register r is decreased by one, followed by a jump to instruction q. The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.

– $l_h : halt$ (HALT instruction)
  Stop the machine. The final label l_h is only assigned to this instruction.

The register machines are known to be computationally complete, equal in power to (non-deterministic) Turing machines: they generate or accept exactly the sets of vectors of non-negative integers which can be generated by Turing machines, and they can even compute any partial recursive relation on vectors of non-negative integers.

For example, a (non-deterministic) register machine M is said to generate a vector $(s_1, \cdots, s_\beta)$ of non-negative integers if, starting with the instruction with label l_0 and all registers containing the number 0, the machine stops (it reaches the instruction $l_h : halt$) with the first β registers containing the numbers $s_1, \cdots, s_\beta$ (and all other registers being empty).

3 Extended Spiking Neural P Systems

The reader is supposed to be familiar with basic elements of membrane computing, e.g., from [9] and [11]; comprehensive information can be found on the

P systems web page [13]. Moreover, for the motivation and the biological background of spiking neural P systems we refer the reader to [6] as well as to the corresponding Chapter 13 in the Handbook of Membrane Computing [11]. For the definition of an *extended spiking neural P system* we refer to [3].

Based on the model of *extended spiking neural P systems*, we now define the new model of *extended spiking neural P systems with states*, i.e., the neurons can be in different states, and depending on the current state of a neuron, different spiking rules may be applicable.

Definition 1. *An* extended spiking neural P system with states *(of degree* $m \geq 1$*) (an* ESNPS system *for short) is a construct* $\Pi = (N, S, I, R, f)$ *where*

- N *is the set of* cells *(or* neurons*); the neurons may be uniquely identified by a number between* 1 *and* m *or by an alphabet of* m *symbols;*
- S *is the set of* states*;*
- I *describes the* initial configuration *by assigning an initial value (of spikes) and an* initial state *to each neuron;*
- R *is a finite set of* rules *of the form* $i : \left(s_i, a^k\right) \rightarrow P$ *such that* $i \in N$ *(specifying that this rule is assigned to neuron* i*),* $s_i \in S$ *is the current state of neuron* i*,* $k \in \mathbb{N}$ *is the "number of spikes" (the energy) consumed by this rule, and* P *is a (possibly empty) set of* productions *of the form* (l, w, s) *where* $l \in N$ *(thus specifying the target neuron),* $w \in \{a\}^*$ *is the* weight *of the energy sent along the axon from neuron* i *to neuron* l*, and* $s \in S$ *is the* state signal *sent along the axon from neuron* i *to neuron* l*;*
- f *is the state composition function, which for each neuron allows for computing the new state of a neuron from its current state and the state signals having arrived in the neuron in the previous step.*

Definition 2. *A* configuration *of the ESNPS system is described by specifying, for each neuron, the actual number of spikes in the neuron as well as its current state. A* transition *from one configuration to another one now works as follows:*

- *for each neuron* i*, we first choose (if possible) one rule* $i : \left(s_i, a^k\right) \rightarrow P$ *which is applicable (this means neuron* i *must be in state* s_i *and contain at least* k *spikes); by applying this rule, we reduce the number of spikes in neuron* i *by* k*; moreover, for each production* $(l, w, s) \in P$*,* $|w|$ *spikes and the state signal* s *are sent to neuron* l*;*
- *for each neuron* l*, we now consider all "packages"* (l, w, s) *having been sent on axons leading to neuron* l*; we then sum up all weights* w *in such packages and add this sum of spikes to the corresponding number of spikes in neuron* l*;*
- *for each neuron* l*, moreover, we use the current state and all the state signals* s *(if any) having arrived in the neuron for computing the next state of the neuron by the state composition function* f*.*

After having executed all the substeps described above in the correct sequence, we obtain the description of the new configuration. A computation *is a sequence of configurations starting with the initial configuration given by* I*. A computation is called* successful *if it halts, i.e., if no neuron contains an applicable rule and*

no neuron would be able to change its state in the next step (thus making other spiking rules applicable). Observe that even if no rule has been applicable, a neuron still might be able to change its state as the state composition function f also uses the current state of the neuron to compute its new state and does not necessarily rely on additional state signals having arrived from other neurons.

With respect to the original model introduced in [6] as well as to the model with polarizations as considered in [14], our model as defined above is more general: the most important extension is that different rules for neuron i may affect different axons leaving from it whereas in these other model the structure of the axons (called synapses there) is fixed. Moreover, reflexive axons, i.e., leading from neuron i to neuron i, are not allowed there, i.e., for (l, w, s) being a production in a rule $i : \left(s_i, a^k\right) \to P$ for neuron i, $l \neq i$ is required. On the other hand, our definitions have been chosen in such a way that the model introduced in [14] is a restricted variant of our more general model.

Finally, we mention that as in [4], the notion of *extended* spiking neural P systems often is used only taking into account that more than one spike can be sent along all axons with one spiking rule.

Depending on the purpose the ESNPS system is to be used, i.e., as a generative, an accepting, or a computing device, some more features have to be specified: for generating k-dimensional vectors of non-negative integers, we have to designate k neurons as *output neurons*, and the other neurons then will also be called *actor neurons*; for the computing case and the accepting case, some neurons have to be designated as *input neurons*. As in [3], also in this paper, we take the number of spikes at the beginning/end of a successful computation in the input/output neurons as the input/output values.

Remark 1. In the following, for a spiking rule $i : \left(s_i, a^k\right) \to P$, the set P will not be written as a set, but just by concatenating its elements of the form (l, w, s), where l is the target neuron, w describes the number of spikes sent to l and s is the state signal sent to l.

The following example illustrates the computational power of ESNPS systems with two states by showing how sets of exponentially growing numbers can be generated.

Example 1. We construct the ESNPS system

$$\Pi_{4^n} = (\{\sigma_1, \sigma_1', \sigma_2, \sigma_2'\}, \{0, 1\}, I, R, f)$$

generating the multiset language $\{a^{4^n} \mid n > 1\}$ in the output neuron σ_1. Initially, σ_1 and σ_1' are in state 1 and contain one and two spikes, respectively. On the other hand, neurons σ_2 and σ_2' initially are in state 0 and are empty.

The state composition function f, for every neuron, is given as follows: If any state signal 1 has arrived in the previous step, the state of the neuron is 1, and 0 otherwise.

To illustrate the rules in the ESNPS system, we use the following graphical notation: Each rule is represented by an arrow with a single tail but with multiple heads; the branching point is highlighted by a black bullet. The left-hand side of the rule is written on the segment preceding the bullet and each right-hand side on the corresponding arrow head. When writing the right-hand sides, we omit the names of the target neurons (because they are pointed at by the arrow heads).

Π_{4^n} works in a two-phase cycle. In the first phase, all the spikes from σ_1 are transferred in two copies into σ_2; this phase is controlled by σ'_1. The second phase is symmetric: the spikes from σ_2 are doubled and moved into σ_1, under the control of σ'_2.

The first phase is governed by the following rules in neurons σ_1 and σ'_1:

$$\begin{aligned}
&\sigma_1 : (1, a) \to (\sigma_2, a^2, 0)(\sigma'_1, a, 1),\\
&\sigma'_1 : (1, a) \to (\sigma_1, \lambda, 1),\\
&\sigma'_1 : (0, a) \to (\sigma_2, \lambda, 1)(\sigma'_2, a^2, 1).
\end{aligned}$$

The graphical representation of these rules is given in Figure 1.

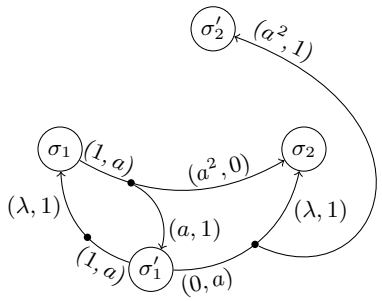


Fig. 1. Multiplication by 2 in ESNPS with two states.

The loop between σ_1 and σ'_1 ensures that, while there are still spikes in σ_1, both neurons stay in state 1. When there are no more spikes in σ_1, σ'_1 must pass into state 0 and will have to use its last spike (of the two it normally contains) to apply the rule $(0, a) \to (\sigma_2, \lambda, 1)(\sigma'_2, a^2, 1)$. This rule puts two spikes into the control neuron σ'_2 and switches both neurons σ_2 and σ'_2 to state 1, thereby starting the second phase of the cycle. The second phase is totally symmetric and is governed by the following rules in neurons σ_2 and σ'_2:

$$\begin{aligned}
&\sigma_2 : (1, a) \to (\sigma_1, a^2, 0)(\sigma'_2, a, 1),\\
&\sigma'_2 : (1, a) \to (\sigma_2, \lambda, 1),\\
&\sigma'_2 : (0, a) \to (\sigma_1, \lambda, 1)(\sigma'_1, a^2, 1).
\end{aligned}$$

Finally, to ensure that the system halts after the second phase of the cycle, we add the following rule to the control neuron σ_2':

$$\sigma_2' : (0, a) \to (\sigma_1, \lambda, 0).$$

Thus, σ_2' may choose between restarting the loop by switching the state of σ_1 and σ_1', or just forgetting the last control spike, thus effectively bringing the system to a halt, with 4^n copies of a in σ_1, where n is the number of times the cycle has run.

4 Simulating Register Machines with Extended Spiking Neural P Systems with only Two States

We now consider an arbitrary *n-register machine* $M = (n, B, l_0, l_h, P)$ and show how to simulate the computations of such a register machine by a spiking neural P system $\Pi = (N, S, I, R, f)$ with only two states, i.e., $S = \{0, 1\}$.

In the initial configuration, except for neuron l_0, all neurons are in state 0 and contain no spike (in the accepting or the computing case, the input neurons contain the input values), and only neuron l_0 contains one spike.

For all neurons, we use one global state composition function, i.e., if any state signal 1 has arrived in the previous step, the state of the neuron is 1, and 0 otherwise.

In the following, we will provide several variants of spiking neural P systems with only two states simulating the computations of a register machine by showing how `ADD`- and `SUB`-instructions can be simulated.

A first simple proof. We start with a very simple proof using the possibilities offered by our rather general definition.

For each register r, $1 \leq r \leq n$, we use a corresponding neuron r. For each instruction of the register machine with label $p \in B$ we use a neuron p and some additional neurons to simulate this instruction.

An `ADD`-instruction $p : (\texttt{ADD}(r), q, s)$ is simulated by neuron p with the rules

$p : (0, a) \to (q, a, 0)(r, a, 0)$ and
$p : (0, a) \to (s, a, 0)(r, a, 0)$,

meaning that neuron p (always staying in state 0) consumes one spike and sends one spike and state 0 to neuron q or neuron s and to neuron r (the neuron representing register r).

For `SUB`-instructions on register r, let

$$SUB(r) = \{p \in B \mid p : (\texttt{SUB}(r), q, s) \in P\}.$$

Then the rule in neuron r representing register r allowing for simulating `SUB`-instructions is

$r : (1, a) \to \prod_{l \in SUB(r)} (l'', \lambda, 1)$.

In case the register is non-empty, in the activated state 1 one spike is eliminated and state 1 is sent to every neuron l'' for every label l of a SUB-instruction, yet no spike a is sent.

Then a SUB-instruction $p : (\mathtt{SUB}(r), q, s)$ is simulated by the neurons p, p', and p'' with the rules

$p : (0, a) \to (p', a, 1)(r, \lambda, 1)$,
$p' : (1, a) \to (p'', a, 0)$, as well as
$p'' : (0, a) \to (s, a, 0)$ and
$p'' : (1, a) \to (q, a, 0)$.

The simple construction described above obeys to the following features, interesting from a complexity point of view:

- as desired, we only need two states;
- we do not use self-loops;
- we send the same state to all neurons in each rule;
- exactly one spike is consumed by each rule;
- yet, on the other hand, we allow to also send *zero* spikes to a neuron.

The connection structure only depends on the state in case of deterministic register machines! Yet by using a more complicated construction for the simulation of non-deterministic ADD-instructions we can even obtain that feature in general:

A refined proof where the connection structure between the neurons only depends on the state. An ADD-instruction $p : (\mathtt{ADD}(r), q, s)$ is simulated by the neurons p, p', and p'' with the rules

$p : (0, a) \to (p', a, 0)(r, a, 0)$,
$p' : (0, a) \to (p'', a, 0)$, and
$p' : (0, a) \to (p'', a, 1)$, as well as
$p'' : (0, a) \to (q, a, 0)$ and
$p'' : (1, a) \to (s, a, 0)$.

A proof with a static connection structure between the neurons. For comparison with the model considered in [14] (where states are called *polarizations*) we have to ask the following question: *Can we have a completely static connection structure even not depending on the state of the neuron?*

We first show that a non-deterministic ADD-instruction can be simulated within a fixed connection structure, now using the initial neuron p in the activated state 1:

An ADD-instruction $p : (\mathtt{ADD}(r), q, s)$ is simulated by the neurons p and p' with the rules

$p : (1, a) \to (p', a, 0)(r, a, 0)$,

$p' : (0, a) \to (0''_q, a, 0)(1''_s, a, 0)$, and
$p' : (0, a) \to (0''_q, a, 1)(1''_s, a, 1)$,
together with the following rules in the neurons $0''_l$ and $1''_l$, for every label $l \in B$:

$0''_l : (0, a) \to (l, a, 1)$ and
$0''_l : (1, a) \to \lambda$ as well as
$1''_l : (1, a) \to (l, a, 1)$ and
$1''_l : (0, a) \to \lambda$.

The rules of the form $p : (s, a) \to \lambda$ with $s \in \{0, 1\}$ (and $p = (1-s)''_l$) are rules usually called *forgetting rules* as they only consume spikes in the neuron p without sending spikes to the connected neuron l; yet with respect to emphasizing a fixed connection structure, they rather could also be written as
$p : (s, a) \to (l, \lambda, 0)$
i.e., although *zero* spikes are sent to neuron l, still the state signal 0 is sent along the axon to l. A careful inspection of our proofs shows that in both interpretations – whether sending the state signal 0 or not –, the simulations work correctly.

Instead of showing how SUB-instructions can also be simulated within a fixed connection structure, we finally attack the last remaining non-standard feature at the same time, i.e.: *Can we avoid sending zero spikes?*

The final proof for the non-extended model. With using the simulation(s) of ADD-instructions as given in the previous proof variant, we obtain a simulation for a model comparable with that one as considered in [14], yet improving the result from three states (polarizations) to only two.

For each register r, $1 \leq r \leq n$, we now use two neurons r and r'.

In the initial configuration, except for neuron l_0, all neurons are in state 0 and contain no spikes (in the accepting or the computing case, the input neurons contain the input values), and only neuron l_0 is in state 1 and contains one spike.

As we now are dealing with a fixed connection structure between the neurons in N, which usually (e.g., see [14]) is represented by a relation $syn \subseteq N \times N$, a rule in a neuron i of the form

$$i : \left(s_i, a^k\right) \to \{(l, a^m, s) \mid (i, l) \in syn\}$$

now will be written as

$$i : \left(s_i, a^k\right) \to (a^m, s) \mid \{j \mid (i, j) \in syn\}\,.$$

This notation specifies the connection structure syn in an implicit way, but allows for an easy non-graphical representation of the ESNPS system.

We first use this new notation to repeat how a non-deterministic ADD-instruction $p : (\texttt{ADD}(r), q, s)$ is simulated by the neurons p and p' with the rules

$p : (1, a) \to (a, 0) \mid \{p', r\}$,
$p' : (0, a) \to (a, 0) \mid \{0''_q, 1''_s\}$ and
$p' : (0, a) \to (a, 1) \mid \{0''_q, 1''_s\}$,

together with the following rules in the neurons $0''_l$ and $1''_l$, for every label $l \in B$:

$0''_l : (0, a) \to (a, 1) \mid \{l\}$ and
$0''_l : (1, a) \to (\lambda, 0) \mid \{l\}$ as well as
$1''_l : (1, a) \to (a, 1) \mid \{l\}$ and
$1''_l : (0, a) \to (\lambda, 0) \mid \{l\}$.

A SUB-instruction $p : (\texttt{SUB}(r), q, s)$ is simulated by the neurons p, $\tilde{p}$, $\tilde{p}'$, $\hat{p}$, $\hat{p}'$, $\hat{p}''$, $\bar{p}$, and $\bar{p}'$ with the rules

$p : (1, a) \to (a, 1) \mid \{\tilde{p}, \hat{p}, r\}$,
$\tilde{p} : (1, a) \to (a, 0) \mid \{\tilde{p}'\}$,
$\tilde{p}' : (1, a^2) \to (a, 1) \mid \{q\} \cup \{\hat{l}'' \mid l \in SUB(r) \setminus \{p\}\}$,
$\tilde{p}' : (0, a) \to (\lambda, 0) \mid \{q\} \cup \{\hat{l}'' \mid l \in SUB(r) \setminus \{p\}\}$ as well as

$\hat{p} : (1, a) \to (a, 1) \mid \{\hat{p}'\}$,
$\hat{p}' : (1, a) \to (a, 0) \mid \{\hat{p}''\}$,
$\hat{p}'' : (0, a) \to (a, 1) \mid \{r, \bar{p}\}$,
$\hat{p}'' : (1, a^2) \to (\lambda, 0) \mid \{r, \bar{p}\}$,
$\bar{p} : (1, a) \to (a, 1) \mid \{\bar{p}'\}$, and
$\bar{p}' : (1, a) \to (a, 1) \mid \{s\} \cup \{\hat{l}'' \mid l \in SUB(r)\}$

together with the following rules for the register neuron r and for the additional neuron r':

$r : (1, a^2) \to (a, 1) \mid \{r'\} \cup \{\tilde{l}' \mid l \in SUB(r)\}$ and

$r' : (1, a) \to (a, 1) \mid \{\hat{l}'' \mid l \in SUB(r)\}$.

The main idea of this construction is to start both decrement and zero-check case in parallel and then, depending on the signal from r and r' take the necessary action, including to *reset* register r to 0 if the additional spike sent there did not lead to a spiking action of neuron r in case the value stored in the register was zero. Moreover, all actor neurons affected by signals from neuron r not belonging to the current label p have to be reset without allowing them to act in a non-desired way:

For the neurons $\tilde{p}'$ this happens *automatically* as with only one spike a they cannot spike in state 1, yet after one step the state goes back to 0 and then allows the spike to be forgotten.

For the neurons $\hat{p}''$ this happens if the state signal 1 and the spike from neuron r' are accompanied by a second spike which allows for resetting the neuron by using the forgetting rule $(1, a^2) \to (\lambda, 0)$.

5 Universal (Extended) Spiking Neural P Systems with Two States

We simulate the strongly universal register machine U_{22} of Korec, see [7]. Rather than performing a direct simulation which would yield $9 \times 1 + 8 \times 1 + 13 \times 4 = 69$ rules, we notice that simulation of ADD-instructions does not require separate rules, because it can be done as a part of the simulation of SUB-instructions. More exactly, increments are built into the transitions to q and s of $p : (\mathtt{SUB}(r), q, s)$. This has been formalized as *generalized register machine* (GRM for short), see [1]. The rules of U_{22} in the GRM form are given below.

$$
\begin{aligned}
&q_1 : (\mathtt{SUB}(1), \mathtt{ADD}(7)q_1, \mathtt{ADD}(6)q_4),\\
&q_4 : (\mathtt{SUB}(5), \mathtt{ADD}(6)q_4, q_7),\\
&q_7 : (\mathtt{SUB}(6), \mathtt{ADD}(5)q_{10}, q_4),\\
&q_{10} : (\mathtt{SUB}(7), \mathtt{ADD}(1)q_7, q_{13}),\\
&q_{13} : (\mathtt{SUB}(6), \mathtt{ADD}(6)q_{14}, q_1),\\
&q_{14} : (\mathtt{SUB}(4), q_1, q_{16})\\
&q_{16} : (\mathtt{SUB}(5), q_{18}, q_{23}),\\
&q_{18} : (\mathtt{SUB}(5), q_{20}, q_{27}),\\
&q_{20} : (\mathtt{SUB}(5), \mathtt{ADD}(4)q_{16}, \mathtt{ADD}(2)\mathtt{ADD}(3)q_{32}),\\
&q_{23} : (\mathtt{SUB}(2), q_{32}, q_{25}),\\
&q_{25} : (\mathtt{SUB}(0), q_1, q_{32}),\\
&q_{27} : (\mathtt{SUB}(3), q_{32}, \mathtt{ADD}(0)q_1),\\
&q_{32} : (\mathtt{SUB}(4), q_1, q_h).
\end{aligned}
$$

We note that also the first step of the simulation of a generalized SUB-instruction can be embedded into the last step of the preceding simulation. Moreover, note that in this case we may start with one spike in neuron q''_{13}. It is easy to see that it suffices to have 3 rules per each of the 13 generalized conditional decrement instructions and 1 rule per each of the 8 registers, yielding a total of only **47** rules, associated to register neurons, primed instruction neurons and double-primed instruction neurons.

Register neurons

$$
\begin{aligned}
&0 : (1, a) \to (q''_{25}, \lambda, 1),\\
&1 : (1, a) \to (q''_{1}, \lambda, 1),\\
&2 : (1, a) \to (q''_{23}, \lambda, 1),\\
&3 : (1, a) \to (q''_{27}, \lambda, 1),\\
&4 : (1, a) \to (q''_{14}, \lambda, 1)(q''_{32}, \lambda, 1),\\
&5 : (1, a) \to (q''_{4}, \lambda, 1)(q''_{16}, \lambda, 1)(q''_{18}, \lambda, 1)(q''_{20}, \lambda, 1),\\
&6 : (1, a) \to (q''_{7}, \lambda, 1)(q''_{13}, \lambda, 1),\\
&7 : (1, a) \to (q''_{10}, \lambda, 1).
\end{aligned}
$$

The **primed instruction neurons** have the rules

$$q_i' : (1, a) \to (q_i'', a, 0) \text{ for } i \in \{1, 4, 7, 10, 13, 14, 16, 18, 20, 23, 25, 27, 32\}.$$

We give the rules of **double-primed instruction neurons** in the table below, the row representing the neuron, the column representing the left side of a rule, and their intersection containing the right side of that rule.

	$(0, a)$	$(1, a)$
q_1''	$(q_4', a, 1)(5, \lambda, 1)(6, a, 0)$	$(q_1', a, 1)(1, \lambda, 1)(7, a, 0)$,
q_4''	$(q_7', a, 1)(6, \lambda, 1)$	$(q_4', a, 1)(5, \lambda, 1)(6, a, 0)$,
q_7''	$(q_4', a, 1)(5, \lambda, 1)$	$(q_{10}', a, 1)(7, \lambda, 1)(5, a, 0)$,
q_{10}''	$(q_{13}', a, 1)(6, \lambda, 1)$	$(q_7', a, 1)(6, \lambda, 1)(1, a, 0)$,
q_{13}''	$(q_1', a, 1)(1, \lambda, 1)$	$(q_{14}', a, 1)(4, \lambda, 1)(6, a, 0)$,
q_{14}''	$(q_{16}', a, 1)(5, \lambda, 1)$	$(q_1', a, 1)(1, \lambda, 1)$,
q_{16}''	$(q_{23}', a, 1)(2, \lambda, 1)$	$(q_{18}', a, 1)(5, \lambda, 1)$,
q_{18}''	$(q_{27}', a, 1)(3, \lambda, 1)$	$(q_{20}', a, 1)(5, \lambda, 1)$,
q_{20}''	$(q_{32}', a, 1)(4, \lambda, 1)(2, a, 0)(3, a, 0)$	$(q_{16}', a, 1)(5, \lambda, 1)(4, a, 0)$,
q_{23}''	$(q_{25}', a, 1)(0, \lambda, 1)$	$(q_{32}', a, 1)(4, \lambda, 1)$,
q_{25}''	$(q_{32}', a, 1)(4, \lambda, 1)$	$(q_1', a, 1)(1, \lambda, 1)$,
q_{27}''	$(q_1', a, 1)(1, \lambda, 1)(0, a, 0)$	$(q_{32}', a, 1)(4, \lambda, 1)$,
q_{32}''	$(q_h, a, 0)$	$(q_1', a, 1)(1, \lambda, 1)$.

This construction uses a total of $8 + 2 \times 13 + 1 = \mathbf{35}$ neurons. The halting neuron q_h can be avoided, for example, by changing the right side of the rule with $q_{32}'' : (0, a)$ to $(4, \lambda, 1)$. Indeed, the register machine halts with register 4 being empty, so the P system will halt after neuron 4 has reset its state to 0. We also remark that this construction does not respect the requirement of all states on the right side being equal. This requirement can be fulfilled by replacing $(r, a, 0)$ by $(r', a, 1)$ for $r \in \{0, 1, 4, 5, 6, 7\}$ and $(2, a, 0)(3, a, 0)$ by $(\langle 2, 3\rangle', a, 1)$ in the right sides of the rules above, and adding 7 additional neurons, each having one rule: $r' : (1, a) \to (r, a, 0)$ for $r \in \{0, 1, 4, 5, 6, 7\}$ and the neuron $\langle 2, 3\rangle$ with the rule $\langle 2, 3\rangle' : (1, a) \to (2, a, 0)(3, a, 0)$, yielding a total of **54** rules.

If we consider the most restricted variant of spiking neural P systems with states elaborated at the end of Section 4, which is comparable with the model considered in [14] using the notion of *polarizations* instead of the notion *states*, when again embedding the first step of the simulation of a generalized SUB-instruction into the last step of the preceding simulation, a straightforward calculation yields two neurons and two rules per register as well as 7 neurons and 9 rules per generalized conditional decrement instruction, which yields a total of $16 + 7 \times 13 = \mathbf{107}$ neurons as well as $16 + 9 \times 13 = \mathbf{133}$ rules.

6 Conclusion

We have shown that only two states (or polarizations as they are called in [14]) are needed for obtaining computational completeness with (extended) spiking

neural P systems with states, thus solving an open problem raised at the Brainstorming Week on Membrane Computing in Sevilla at the beginning of February 2016.

As interesting variants for future research we suggest to investigate the influence of how to choose the state composition function, e.g., what changes if we use other global functions; moreover, it may also be interesting to have different local functions instead of one global function, especially functions also taking into account the current state of the neuron.

References

1. A. Alhazov and R. Freund. Variants of small universal P systems with catalysts. *Fundamenta Informaticae*, 138(1–2):227–250, 2015.
2. A. Alhazov, R. Freund, S. Ivanov, M. Oswald, and S. Verlan. Extended spiking neural P systems with white holes. In L. Macías-Ramos, Gh. Păun, A. Riscos-Núñez, and L. Valencia-Cabrera, editors, *Proceedings of the 13th Brainstorming Week on Membrane Computing*, pages 45–62. Fénix Editora, Sevilla, 2015.
3. A. Alhazov, R. Freund, M. Oswald, and M. Slavkovik. Extended spiking neural P systems. In *Workshop on Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2006.
4. H. Chen, M. Ionescu, T.-O. Ishdorj, A. Păun, Gh. Păun, and M. Pérez-Jiménez. Spiking neural P systems with extended rules: universality and languages. *Natural Computing*, 7(2):147–166, 2007.
5. R. Freund and M. Oswald. A short note on analysing P systems. *Bulletin of the EATCS*, 78:231–236, 2002.
6. M. Ionescu, Gh. Păun, and T. Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2,3):279–308, 2006.
7. I. Korec. Small universal register machines. *Theoretical Computer Science*, 168:267–301, 1996.
8. M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
9. Gh. Păun. *Membrane Computing: An Introduction*. Natural Computing Series Natural Computing. Springer, 2002.
10. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 2003.
11. Gh. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
12. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1-3. Springer, 1997.
13. The P Systems Website. http://ppage.psystems.eu.
14. T. Wu, A. Păun, Z. Zhang, and L. Pan. Spiking neural P systems with polarizations. *Submitted*, 2015.

Selection Criteria for Statistical Model Checking

Mehmet E. Bakir and Mike Stannett*

Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK

Abstract. Statistical model checking (SMC) has been used to verify both biological and P systems, but different SMC tools employ different modelling and property specification languages, making it hard to decide which tool is best for which problem. We survey the capabilities of SMC tools and provide experimental results showing their ability to verify patterns against biological models. Our eventual goal is the automation of the SMC selection process.

1 Introduction

Simulation and model checking have been applied to various computational models, including P systems. Although simulation is relatively fast and computationally inexpensive, there is no guarantee that all computational paths will be examined. In contrast, model checking considers all possible paths in order to guarantees the correctness of model properties, but this can be prohibitively expensive computationally. *Statistical model checking* (SMC) integrates the two techniques by generating and verifying a number of simulation paths to determine an "approximate correctness" measure using statistical methods. This enables faster verification of large models, within specified confidence bounds.

2 Comparison of Statistical Model Checking Tools

Current SMCs use different modelling and specification languages, and support the specification of different property types, so practitioners need to know *in advance* which SMC is best suited to their problem. Here we review the modelling and specification languages of the most widely used SMC tools: PRISM,[1] PLASMA-Lab,[2] Ymer,[3] MRMC[4] and MC2[5] (*see* Table 1). While PRISM and MRMC allow both numerical and statistical checking, the others support only SMC. PLASMA-Lab and Ymer both have reasonable support for the PRISM language, but MRMC and MC2 require users to learn external tools as well, because they do not employ a high-level modelling language.

* The authors would like to thank the referees for their many helpful suggestions.

[1] `http://www.prismmodelchecker.org/`
[2] `https://project.inria.fr/plasma-lab/`
[3] `http://www.tempastic.org/ymer/`
[4] `http://www.mrmc-tool.org/`
[5] `http://www.brunel.ac.uk/research/research-areas/research-groups/cssb/software-systems-and-databases/mc2`

Table 1. Modelling languages and external dependency of SMC tools.

SMCs	Methods	Modelling Language	Needs an External Tool?	External Tool Modelling Language
PRISM	Numerical and Statistical model checking	Probabilistic Reactive Modules, a.k.a, PRISM language	NO	N/A
PLASMA-Lab	Statistical model checking	Reactive Modules Language (RML) of PRISM, Adaptive RML (extension of RML for adaptive systems), RML with importance sampling, Biological Language	NO	N/A
Ymer	Statistical model checking	PRISM language	NO	N/A
MRMC	Numerical and Statistical model checking	Transition matrix	YES, e.g., PRISM	PRISM language
MC2	Statistical model checking	N/A	YES, e.g., Gillespie2	SBML

Key. Not Applicable (N/A), Systems Biology Markup Language (SBML).

Model checkers use temporal logics as property specification languages, so to ease the specification process for non-logicians various recurring properties (*patterns*) have been categorized by previous studies [1, 3]. Table 2 lists various popular patterns with their temporal logic formulations. The table compares the expressibility of these specification languages, showing whether properties can be defined using just one temporal logic operator ("directly supported"), require a more complicated combination of such operators ("indirectly supported"), or are not supported at all.

Table 2. Expressibility of property patterns

Patterns	Description	Temporal Logic	PRISM	PLASMA -Lab	Ymer	MRMC	MC2
Existence	ϕ_1 will eventually hold, within the $\bowtie p$ bounds.	$P_{\bowtie p}[F\ \phi_1]$ or $P_{\bowtie p}[true\ \mathrm{U}\ \phi_1]$	DS	DS	DS	DS	DS
Until	ϕ_1 will hold continuously until ϕ_2 eventually hold, within the $\bowtie p$ bounds.	$P_{\bowtie p}[\phi_1\ \mathrm{U}\ \phi_2]$	DS	DS	DS	DS	DS
Response	If ϕ_1 holds, then ϕ_2 must hold within the $\bowtie p$ bounds.	$P_{\geq 1}[G\ (\phi_1 \rightarrow (P_{\bowtie p}[F\ \phi_2]))]$	**NS**	IS	**NS**	IS	IS
Steady-State (Long-run)	In the long-run ϕ_1 must hold, within the $\bowtie p$ bounds.	$S_{\bowtie p}[\phi_1]$ or $P_{\bowtie p}[FG\ (\phi_1)]$	**NS**	IS	**NS**	DS	IS
Universality	ϕ_1 continuously holds, within the $\bowtie p$ bounds.	$P_{\bowtie p}[G\ \phi_1]$ or $P_{\bowtie (1-p)}[(F\ (\neg\phi_1)]$	DS	DS	IS	DS	DS

Key. ϕ_1, and ϕ_2 are state formulas; $\bowtie \in \{<, >, \leq, \geq\}$; $p \in [0, 1]$ is a probability with rational bounds; and $\bowtie$ is negation of $\bowtie$. $P_{\bowtie p}$ is the *qualitative* operator which enables users to query qualitative features, to query *quantitative* properties, $P_{=?}$ (quantitative operator) can be used. DS = Directly Supported, IS = Indirectly Supported, NS = Not Supported.

The Existence, Until and Universality patterns are directly supported by all 5 tools, with the exception that Ymer requires Universality to be given indirectly.

Table 3. SMC capabilities: the number of model/pattern verifications completed by each SMC tool within an hour.

SMCs	Existence	Until	Response	Steady-State	Universality
PRISM	337	435	**N/A**	**N/A**	370
PLASMA-Lab	465	465	465	465	465
Ymer	439	439	**N/A**	**N/A**	439
MRMC (with PRISM)	75	72	75	57	77
MC2 (with Gillespie2)	458	458	458	458	458

Notes. Model selected from the BioModels database [2] (as modified by the authors of [4]), their sizes ranges from 2 species and 1 reaction to 2631 species and 2824 reactions. The models were verified against five patterns: Existence, Until, Response, Steady-State and Universality. The experiments are conducted on Intel i7-2600 CPU @ 3.40GHz 8 cores, with 16GB RAM running on Ubuntu 14.04.

The Response pattern is not supported by PRISM or Ymer, and is only indirectly supported by PLASMA-Lab, MRMC and MC2. The Steady-State pattern can be either represented by one operator, S, or two operators, F and G. Only MRMC allows Steady-State to be specified with one operator (S), while PLASMA-Lab and MC2 allow it to be expressed indirectly.

To determine the relative capabilities of each SMC tool, we verified 465 biological models against five patterns [3]. Table 3 shows the number of models that each SMC tool was able to verify within an hour (integrated, where necessary, with external supporting tools).

PLASMA-Lab was the only tool that verified all models 465 models in time. MC2 (with Gillespie2) was able to verify 458 models, and Ymer could verify 439 model against all *supported* patterns, but timed-out for the others. PRISM was able to verify 337, 435 and 370 models against Existence, Until and Universality, respectively. PRISM failed to verify the remained models since higher path depth required. MRMC was able to verify only a small number of models, because it relies on PRISM which crashed for large models before matrix exportation .

Our experiments confirm that the capabilities of SMC tools differ not just intrinsically, but that variations depend on the properties being queried. Further investigation is, therefore, under way to determine how best to automate the SMC selection process.

References

1. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-state Verification. In: Proc. 21st Int. Conf. on Softw. Eng. pp. 411–420. ICSE '99, ACM, New York, NY, USA (1999)
2. European Bioinformatics Institute. `http://www.ebi.ac.uk/`, accessed 18/04/16
3. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proc. 30th Int. Conf. Softw. Eng. pp. 31–40. ICSE '08, ACM, New York, NY, USA (2008)
4. Sanassy, D., Widera, P., Krasnogor, N.: Meta-stochastic simulation of biochemical models for systems and synthetic biology. ACS Synthetic Biology 4(1), 39–47 (2015)

Towards Agent-Based Simulation of Kernel P Systems using FLAME and FLAME GPU

Raluca Lefticaru[1,2], Luis F. Macías-Ramos[3],
Ionuţ Mihai Niculescu[4], Laurenţiu Mierlă[2,4]

[1] CENTRIC, Sheffield Hallam University,
153 Arundel Street, Sheffield, S1 2NU, UK
[2] Department of Computer Science, University of Bucharest,
Str. Academiei nr. 14, 010014, Bucharest, Romania
raluca.lefticaru@gmail.com
[3] Research Group on Natural Computing, Dept. Computer Science and Artificial Intelligence, University of Seville Avda. Reina Mercedes S/N, 41012, Sevilla, Spain
lfmaciasr@us.es
[4] Department of Mathematics and Computer Science, University of Pitesti,
Str. Targu din Vale 1, 110040, Pitesti, Romania
ionutmihainiculescu@gmail.com, laurentiu.mierla@gmail.com

Abstract. This position paper discusses the challenges and opportunities of simulating kernel P systems (kP systems) using two powerful agent-based modelling frameworks on parallel architectures: FLAME (Flexible Large Scale Agent Modelling Environment) and FLAME GPU, its extension for High Performance Computing (HPC) platforms based on general purpose graphic processing units. These template-based simulation environments have been successfully used in several computationally demanding applications, ranging from macroeconomics to biological systems, thus efforts were put in translating P system models into Communicating Stream X-Machines (CSXM), the core theoretical computational devices of FLAME and FLAME GPU. Following this, translation into FLAME agents for kernel P systems having only rewriting and communication rules was proved always possible and a translation tool integrated in kPWorkbench, the software simulator for kernel P systems, was implemented. However, there are other useful features of kP systems, such as the presence of structure changing rules, like membrane division or dissolution, link creation or destruction rules, that could be taken into account, in order to exploit the full expressiveness and dynamic power of these models.
Keywords: Membrane computing; kernel P systems; communicating stream X-machines; agent-based simulation.

1 Motivation

Kernel P systems (or *kP systems*) are a novel variant of membrane systems aiming to bring together relevant features from several P systems flavours into a unified *kernel* model which allows solving complex problems using a straightforward *code programming approach* [3]. Since P systems in general, and kP systems

in particular, have not been implemented neither *"in vivo"* nor *"in vitro"* nor *"in silico"* yet, a simulation software suite named *kPWorkbench* [4] has been developed to enable specification, parsing and simulation of kP systems models defined in the kernel P–Lingua (*kP–Lingua)* programming language. On the theoretical side, several relevant results involving kP systems have been obtained. Remarkably, it has been also shown that any computation of a kP system involving only rewriting and communication rules can be simulated by a family of Communicating Stream X-Machines (*CSXM*), the core of FLAME [6] *agent based simulation framework*.

Following this, kPWorkbench enables translating kP systems specified in the kP–Lingua language to FLAME models, which allows specification of CSXM to be simulated in a serial or parallel (MPI based) way, by using the FLAME simulation framework [8]. In relation to the FLAME software, a new research project was initiated to develop the FLAME GPU framework [9], aiming to enable the efficient simulation of CSXM on CUDA enabled GPGPU devices.

In this work, we take a step forward regarding the aforementioned results tackling new challenges. Firstly, we address an extension of the kPWorkbench framework to generate FLAME models from kP–Lingua specifications including structural rules such as division and dissolution rules. Secondly, we address the translation of FLAME specifications into the FLAME GPU language.

2 Related Work

FLAME and FLAME GPU have been used in several experiments required to be performed on HPC devices due to the scale of the associated models, e.g. modelling oxygen-responsive transcription factors in *Escherichia coli* [1] or complex cellular tissue simulation [7].

An important theoretical result proving the possibility of simulating kP systems with communication and rewriting rules with CSXM is given in [6]. This was followed by some initial experiments, such as simulating in FLAME a kP system corresponding to a synthetic biology pulse generator [2] and, finally, adding a translator from kP systems with communication and rewriting rules to FLAME specification language into kPWorkbench [4].

One recent work presents a first approach of implementing the pulse generator model in FLAME GPU [5] and conducting a performance comparison with FLAME. However, the model used in [5] was manually translated, since there is no public available tool to automate the conversion to FLAME GPU.

3 Challenges

Two are the goals we pursue in this work. The first one, to tackle the *automatic* translation of kP systems with structural rules into FLAME modelling language. The second one, to address the translation of FLAME models to FLAME GPU specification language. Accomplishing these goals involves several challenges, as we outline in what follows.

Regarding kP systems vs. FLAME, we extended the FLAME conceptual model to incorporate structural changing rules, the most challenging being the implementation of division rules. Among other technical issues, additional agent types were required in order to manage the newly created agents, allocate unique identifiers for them, etc. Subsequently, kPWorkbench translation tool has been extended to incorporate the automatic generation of FLAME models from kP systems with division rules. Finally, we conducted experiments to check our work by performing a comparative study of the simulation results in kPWorkbench of a set of kP systems test models and their translated FLAME counterparts, respectively. With respect to FLAME, besides the serial simulation of the models, which was addressed in previous works like [2], also the parallel MPI based simulation provided by FLAME was considered. Since FLAME does not support MPI simulation in Windows environments, the Sevilla HPC Server [10] `mulhacen` was configured with FLAME and Open MPI [11] to conduct the experiments.

Regarding FLAME vs. FLAME GPU, we have to address the fact that, although FLAME GPU is an extension of FLAME, the models designed for FLAME are not supported by FLAME GPU. Firstly, in FLAME GPU memory is pre-allocated. Consequently, agents memory supports neither dynamic arrays nor even fixed arrays with complex types. As such, the envisioned workaround is serialization of dynamic arrays into static arrays of basic types, recommended to have fixed length equal to a power of 2 according to FLAME GPU specifications. Secondly, while in FLAME it is possible to add several new agents in one step, e.g. when a membrane is divided into 3 new compartments, in FLAME GPU only one agent is possible to be added per function. Consequently, the workaround here implies adding additional functions in the CSXM structure to handle addition of the remaining membranes to be created. Finally, as mentioned in [5], in FLAME GPU each agent can only create a single message, but in case of communication rules, multiple compartments should receive different objects. The workaround here is to expand the memory space for each message, allowing it to contain data for multiple targets.

4 Conclusions and Future Work

In this paper we have presented recent efforts towards translation and simulation of kP systems models using FLAME and FLAME GPU. We have addressed the automatic generation of FLAME specifications from kP–Lingua models by kPWorkbench, as well as the challenges of translating FLAME specifications to FLAME GPU due to constraints of the latter programming model. An extended version of this paper is under preparation, tackling the translation to FLAME GPU and comparisons with FLAME, based on the theoretical study conducted here.

Acknowledgements

The work of RL, IN and LM was supported by the Romanian National Authority for Scientific Research CNCS-UEFISCDI grant PNII-ID-PCE-2011-3-0688.

References

[1] H. Bai, M. D. Rolfe, W. Jia, S. Coakley, R. K. Poole, J. Green, and M. Holcombe. Agent-based modeling of oxygen-responsive transcription factors in *Escherichia coli*. *Plos computational biology*, 10(4), 2014.

[2] M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High Performance Simulations of Kernel P Systems. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ ICESS 2014, Paris, France, August 20-22, 2014*. IEEE, 2014, pp. 409–412.

[3] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierla, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez. Kernel P Systems - Version I. *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*:97–124, Aug. 2013.

[4] M. Gheorghe, S. Konur, F. Ipate, L. Mierla, M. E. Bakir, and M. Stannett. An Integrated Model Checking Toolset for Kernel P Systems. In *Membrane Computing - 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*. G. Rozenberg, A. Salomaa, J. M. Sempere, and C. Zandron, editors. Vol. 9504. In Lecture Notes in Computer Science. Springer, 2015, pp. 153–170.

[5] S. Konur, M. Kiran, M. Gheorghe, M. Burkitt, and F. Ipate. Agent-Based High-Performance Simulation of Biological Systems on the GPU. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. IEEE, 2015, pp. 84–89.

[6] I. Niculescu, M. Gheorghe, F. Ipate, and A. Stefanescu. From Kernel P Systems to X-Machines and FLAME. *Journal of Automata, Languages and Combinatorics*, 19(1-4):239–250, 2014.

[7] P. Richmond, D. C. Walker, S. Coakley, and D. M. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[8] FLAME website. URL: `http://www.flame.ac.uk/`.

[9] FLAME GPU website. URL: `http://www.flamegpu.com/`.

[10] The Sevilla HPC Server. URL: `http://www.gcn.us.es/gpucomputing/`.

[11] The Open MPI project. URL: `https://www.open-mpi.org/`.