Preface

This volume contains the papers presented at the 12th International Conference on Membrane Computing (CMC12, http://cmc12.lacl.fr/) which took place in Fontainebleau, France, in the period 23–26 August 2011.

The first three workshops on membrane computing were organized in Curtea de Argeș, Romania – they took place in August 2000 (with the proceedings published in Lecture Notes in Computer Science, volume 2235), in August 2001 (with a selection of papers published as a special issue of Fundamenta Informaticae, volume 49, numbers 1–3, 2002), and in August 2002 (with the proceedings published in Lecture Notes in Computer Science, volume 2597). The next six workshops were organized in Tarragona, Spain (in July 2003), Milan, Italy (in June 2004), Vienna, Austria (in July 2005), Leiden, The Netherlands (in July 2006), Thessaloniki, Greece (in June 2007), and Edinburgh, UK (in July 2008), with the proceedings published in Lecture Notes in Computer Science, by Springer, as volumes 2933, 3365, 3850, 4361, 4860, and 5391, respectively. The 10th workshop returned to Curtea de Argeș in August 2009 (LNCS volume 5957).

From then on, the workshop became a conference and the series of meetings on membrane computing continued as the Conference on Membrane Computing, with the 2010 edition, CMC11, held in Jena, Germany (LNCS volume 6501). The twelfth edition of the meeting, CMC12, was organized in Fontainebleau, France by the Laboratoire d’Algorithmique Complexité et Logique of the University of Paris Est – Créteil Val de Marne.

The invited speakers for CMC12 were: Artiom Alhazov (Chișinău, Moldova and Milan, Italy), Jacob Beal (Cambridge, MA, USA), Jean-Louis Giavitto (Paris, France), Radu Nicolescu (Auckland, New Zealand) and György Vaszil (Budapest, Hungary). Extended abstracts of these talks are included in this volume.

This volume also incorporates the 27 accepted papers and 4 extended abstracts of the poster presentations. Each of them was subject of at least three referee reports. The Program Committee consisted of 21 members: Artiom Alhazov (Chișinău, Moldova and Milan, Italy), Gabriel Ciobanu (Iasi, Romania), Erzsébet Csuhaj-Varju (Budapest, Hungary), Rudolf Freund (Vienna, Austria), Pierluigi Frisco (Edinburgh, UK), Marian Gheorghe (Sheffield, UK) – co-chair, Thomas Hinze (Jena, Germany), Oscar H. Ibarra (Santa Barbara, USA), Kamala Krithivasan (Chennai, India), Alberto Leporati (Milan, Italy), Vincenzo Manca (Verona, Italy), Maurice Margenstern (Metz, France), Radu Nicolescu (Auckland, New-Zealand), Marion Oswald (Budapest, Hungary), Linqiang Pan (Wuhan, China), Gheorghe Păun (Bucharest, Romania and Seville, Spain), Mario J. Perez-Jimenez (Seville, Spain), Dario Pescini (Milan, Italy), Francisco J. Romero-Campero (Sevilla, Spain), Petr Sosík (Opava, Czech Republic) and Sergey Verlan (Paris, France) – co-chair. It was assisted in the selection process by nine additional reviewers: Oana Agrigoroaiei (Iasi, Romania), Bogdan Aman
The Organizing Committee consisted of Patrick Cégielski (co-chair), Cătălin Dima, Frédéric Gervais, Pierre Konopacki, Olivier Michel, Antoine Spicher, Pierre Valarcher and Sergey Verlan (co-chair).

Continuing the tradition started at the previous edition, the meeting was accompanied by two international satellite workshops: Chemical IT: In-silico Meets In-vitro and the 5th Workshop on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC). Additionally a poster session took place, providing an opportunity of attracting presentations of late-breaking or preliminary results from young researchers and students new in the field.

We gratefully acknowledge the financial support accorded to CMC12 by the University of Paris Est Créteil – Val de Marne, IUT de Sénart-Fontainebleau, Laboratoire d’Algorithme Complexité et Logique, Faculté des Sciences et Technologies of the University of Paris Est Créteil and Agence Nationale de Recherche (project SYNBIOTIC). We also thank the administration of IUT de Fontainebleau for the perfect infrastructure made available to CMC12. Finally, we would like to thank our secretaries Nathalie Gillet and Flore Tsila for the extensive assistance in many issues related to CMC12.

The editors warmly thank the Program Committee, the invited speakers, the authors of the submitted papers, the reviewers, and all the participants.
## Contents

### Invited talks

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties of Membrane Systems</td>
<td>3</td>
</tr>
<tr>
<td><em>Artiom Alhazov</em></td>
<td></td>
</tr>
<tr>
<td>Bringing Biology and Engineering Together with Spatial Computing</td>
<td>15</td>
</tr>
<tr>
<td><em>Jacob Beal</em></td>
<td></td>
</tr>
<tr>
<td>The Modeling and the Simulation of the Fluid Machines of Synthetic Biology</td>
<td>17</td>
</tr>
<tr>
<td><em>Jean-Louis Giavitto</em></td>
<td></td>
</tr>
<tr>
<td>Parallel and Distributed Algorithms in P Systems</td>
<td>33</td>
</tr>
<tr>
<td><em>Radu Nicolescu</em></td>
<td></td>
</tr>
<tr>
<td>Variants of Distributed P Automata and the Efficient Parallelizability of Languages</td>
<td>43</td>
</tr>
<tr>
<td><em>György Vaszil</em></td>
<td></td>
</tr>
</tbody>
</table>

### Regular Papers

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantitative Causality in Membrane Systems</td>
<td>53</td>
</tr>
<tr>
<td><em>Oana Agrigoroaiei, Gabriel Ciobanu</em></td>
<td></td>
</tr>
<tr>
<td>Circular Post Machines and P Systems with Exo- Insertion and Deletion</td>
<td>63</td>
</tr>
<tr>
<td><em>Artiom Alhazov, Alexander Krassovitskiy, Yurii Rogozhin</em></td>
<td></td>
</tr>
<tr>
<td>A Spiking Neural P system simulator based on CUDA</td>
<td>77</td>
</tr>
<tr>
<td><em>Francis Cabarle, Henry Adorna, Miguel A. Martínez–del–Amor</em></td>
<td></td>
</tr>
<tr>
<td>Modularity in P Colonies with Checking Rules</td>
<td>93</td>
</tr>
<tr>
<td><em>Luděk Cienciala, Lucie Ciencialová, Miroslav Langer</em></td>
<td></td>
</tr>
<tr>
<td>A Connection Between Finite dP Automata and Multi-head Finite</td>
<td>109</td>
</tr>
<tr>
<td>Automata</td>
<td></td>
</tr>
<tr>
<td><em>Erzsébet Csalaj–Varjú, György Vaszil</em></td>
<td></td>
</tr>
<tr>
<td>An Adaptive Algorithm for P System Synchronization</td>
<td>127</td>
</tr>
<tr>
<td><em>Michael J. Dinneen, Yun-Bum Kim, Radu Nicolescu</em></td>
<td></td>
</tr>
<tr>
<td>P systems with active membranes operating under minimal parallelism</td>
<td>153</td>
</tr>
<tr>
<td><em>Pierluigi Frisco, Gordon Govan</em></td>
<td></td>
</tr>
</tbody>
</table>
Morphological Algorithms: Membrane Receptor- ligand Interactions and Rule-based Molecule Graph Evolution for Exact Set Cover Problem . 169
   Gerd Gruenert, Gabi Escuela, Peter Dittrich, Thomas Hinze

Chemical Analog Computers for Clock Frequency Control Based on P Modules ........................................... 191
   Thomas Hinze, Christian Bodenstein, Benedict Schau, Ines Heiland, Stefan Schuster

Evolutionary Design of a Simple Membrane System .......................... 213
   Xiaoli Huang, Ge Xiang Zhang, Florentin Ipate

Periodicity as a Dynamical Aspect of Generative Spiking Neural P Systems 225
   Genn Neil Ibo, Henry Adorna

Formal Verification of P Systems with Active Membranes through Model Checking ........................................... 241
   Florentin Ipate, Raluca Lefticaru, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, Cristina Tudose

Basic Concurrency Resolution in Clock-free P Systems ......................... 253
   Sergiu Ivanov

On the Efficiency of Tile Pasting P System with Active Membranes ...... 271
   S. Jebasingh, T. Robinson, Atulya K. Nagar

Matrix Representation of Spiking Neural P Systems with Delay ....... 283
   Kamala Krithivasan, Ajeezh Ramanujan

Asynchronous Extended Spiking Neural P Systems with Astrocytes ...... 299
   Linqiang Pan, Jun Wang, Hendrik Jan Hoogeboom

Simulation of P systems with string objects ........................................ 315
   Veaceslav Macari, Galina Magariu, Tatiana Verlan

A P–Lingua based Simulator for Spiking Neural P Systems ............. 323

Computing with Multi-Membranes ........................................... 347
   Vincenzo Manca, Rosario Lombardo

A methodology based on MP theory for gene expression analysis ...... 365
   Luca Marchetti, Vincenzo Manca

Simulation of Spiking Neural P Systems Using Pnet Lab ................. 381
   Venkata Padmavati Metta, Kamala Krithivasan, Deepak Gary
Contents

Generalized Gandy-Păun-Rozenberg machines ........................................... 395
Adam Obtułowicz

Sequentiality Induced by Spike Number in SNP Systems: small universal machines .......................................................... 409
Andrei Păun

Population P Systems with Moving Active Cells ................................. 421
Isidora Petreska, Petros Kefalas

P systems simulating oracle computations ........................................... 433
Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, Claudio Zandron

P Systems with Chained Rules .............................................................. 447
Dragoș Sburlan

Modelling, testing and verification of P systems with active membranes using Rodin and ProB ................................................. 459
Adrian Tucaru, Florentin Ipate

Abstracts of Poster Presentations

Joining bio-inspired and quantum approaches in computer algebra and computer linguistics .................................................. 471
Artiom Alhazov, Liudmila Bartseva, Sevtlana Cojocaru, Alexander Colesnicov, Ludmila Malahov, Yurii Rogozhin

Creating a P system from a Trace using Genetic Algorithms ............... 479
Alex Ciobanu, Florentin Ipate

A computational model to study the dynamics of Pyrenean Newt (Calotriton asper) ................................................................. 485
Maria Angels Colomer, Albert Montori, Isart Gaspa, Cristian Fondevilla

Plant communities dynamics model using P systems .......................... 497
Cristian Fondevilla, Maria Angels Colomer, Federico Fillat, Luis Valencia-Cabrera

List of Authors ..................................................................................... 509
Invited talks
Properties of Membrane Systems

Artiom Alhazov\textsuperscript{1,2}

\textsuperscript{1} Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academi 5, Chișinău MD-2028 Moldova
E-mail: artiom@math.md

\textsuperscript{2} Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
E-mail: artiom.alhazov@unimib.it

Abstract. The goal of this survey is to overview some important properties of membrane systems that are referred to as “promising” or dynamic. It is sometimes quite desirable to achieve them, although they are mostly undecidable. We summarize how some of these properties affect the computational power or the descriptional complexity of membrane systems. A number of variants of the dynamic systems is also discussed.

1 Introduction

Membrane systems, also called P systems, are a framework of (bioinspired) computational models, see [39], [41] and [45]. In this paper we only consider the models with symbol objects; in this case membrane computing is distributed parallel multiset processing. Generally, a computation is a sequence of transitions between the configurations. Configurations are multisets distributed over a structure (graph or tree), and the transitions are induced by rules, defined by reactants, products and control (additional applicability conditions, if any), viewed as formal computational systems (generating/accepting numeric/vector sets or languages, or computing functions).

Since 1998, a multitude of models and variants have been introduced. We now attempt to give their brief taxonomy. We will not separate membrane systems with symbol objects in different models, since it is in principle possible to combine them: nearly all types of rules are described in terms of object rewriting and communications, sometimes with modifying the underlying structure, see also [32]. Specifying all the definitions is out of the scope of this overview; later we focus on the dynamic properties. Note: this paper is targeted to researchers familiar with membrane computing; it speaks about rather many concepts and models, and defining all of them would make the paper unreasonably large.

Derivation mode. By far the most studied derivation mode is maximal parallelism. Other modes have been considered, e.g., sequential mode, maximal strategy, minimal parallelism, asynchronous mode, lookahead mode.
Much attention has been paid to syntactic properties; they are not only decidable, but whether they hold can be answered “just by looking at the system”. We emphasize the properties describing cooperativeness and control of the systems (later we also look at other syntactic properties, e.g., strong determinism and strong reversibility).

Cooperativeness Objects can change by interacting with other objects, the following may be used: no cooperation, catalysts, multi-stable catalysts. Objects can move by interacting by other objects, using, e.g., targets, uniport, symport/antiport, protons, conditional uniport. Objects can change and/or move by interacting with membranes, using polarizations or not.

Control The applicability of rules may be controlled on presence of objects (promoters, inhibitors, activators) or structure conditions (permeability, polarizations). Alternatively, objects can be controlled by switching the context, by membrane dissolution. A different way of control is changing the set of applicable rules (GP systems, inhibiting/de-inhibiting, polymorphism).

Descriptional complexity is a common approach to further classification of membrane system has been done by considering e.g., number of membranes/cells, weight of rules, number of catalysts/protons, cardinality of the alphabet, and even the number of rules (per membrane or in total).

Finally, a very important category is whether P systems satisfy different properties, called “promising” or dynamic. We now look at them thoroughly.

2 Dynamic properties

We would call a property dynamic if it depends on the behaviour of a system and cannot be easily derived from its description (as opposed to the syntactic properties). Given any finite computation, we assume that the property is easily verifiable. The two usual sources of undecidability are a) that we do not always know whether we deal with finite or infinite computations, and b) that some properties are defined on infinite number of computations (due to non-determinism, to the initial input or to some other parameter).

Of course, we are only interested here in such properties that yield meaningful and motivated restrictions of the class of certain P systems (e.g., the property “multiplicities of all objects are always different” is well-defined, but the corresponding subclass of P systems does not seem interesting). By far the most important and the most studied dynamic property of P systems is determinism.

Definition 1. A P system is deterministic if for every (reachable) configuration there is at most one transition from it.

A slightly more relaxed version of this property is “confluent after each step”, i.e., the next configuration must be unique, not the transition.

Deterministic systems are often shown to be computationally complete acceptors, so it follows that determinism is undecidable for corresponding classes.
Recently in [10] one considered a stronger property, so-called strong determinism, which means that existence of at most one transition is required from each configuration (even from unreachable ones). This detail makes a great difference: strong determinism is decidable and even syntactic, because every pair of rules can be checked for conflict easily. Depending on the control used, strong determinism may be computationally very weak or still computationally complete.

Strong determinism (and strong reversibility, see below) has only been formally introduced for P systems with one working region. In general, the concept might depend on how to understand “all configurations”. In case of a larger static structure, a natural generalization of the one-region definitions is that a configuration is defined by multisets in the nodes of the given structure, and all the shown results would hold. In general, particularly if the structure is not static, a configuration may mean any structure with multisets in its nodes, but it seems that such a property is too restricted.

Reversibility is, essentially, backwards determinism. It tells whether the previous configuration (reachable or not) is unique for every (reachable) configuration. In some cases, reversible systems are still universal, and reversibility is undecidable. (It is not difficult to see that demanding the uniqueness of only reachable previous configurations would lead to a degenerate property: any deterministic system can be converted to an equivalent weakly reversible system.)

Another interesting restriction is the strong reversibility: the uniqueness of the previous configuration is required for all (even non-reachable) configurations. Unlike reversibility, strong reversibility has been shown to be decidable [33].

Always halting is a notion that is sometimes quite desired, but seldom discussed explicitly. For deterministic systems without input, this is simply the halting property, which is a famous undecidable property. For non-deterministic systems, particularly those solving some “practical” problems, one may wish that every computation reaches halting, and thus can be interpreted as yielding some result. Clearly, if a generative system has this property, then it only generates finite sets. Accepting systems are not computationally complete, but the power of recursive sets is clearly enough for the practical purposes.

For systems with input, deterministic or not, always halting would also mean on-every-input. For instance, uniform solution of various (e.g., NP hard) problems by P systems with active membranes are specified by distinguishing the input subalphabet and the input region; notice, however, that valid inputs may be narrower than all multisets over the input subalphabet.

Confluence is a relaxed variant of always halting determinism. Non-deterministic branching is allowed, but all computations must halt with the same result. Confluence may be related to a “practical” desire to obtain a meaningful answer to some underlying problem from a single computation. A “stronger” confluence is when the halting configurations must be identical, not just the result.

A less restricted property is ultimate confluence. It is even allowed for the computations to never halt, but if a system may halt, then the halting
configuration (it follows that it is unique) must be reachable from any reachable configuration. Ultimate confluence may be related to halting with probability 1, in the same configuration.

*Time freeness* is a different kind of dynamic properties. First, (even in maximal parallelism) the times of rule execution (appearance of products of the rule) is not necessarily one step, see also [24]. Second, it is required that the result of (all computations of) a system does not depend on rule execution timings.

One also attempts to define clock freeness as a stronger variant of time freeness. Although not requiring that different executions of the same rule take the same time usually does not affect the proofs, the notion becomes quite complicated. These may be intuitively related to asynchronous termination of reactions.

Compare time freeness with asynchronous P systems. Asynchronous mode can be modeled by maximally parallel systems if for every object $a$ in the alphabet there is a rule $a \rightarrow a$. In this way, objects enter reactions asynchronously.

Finally, one could speak about Petri-net like dynamic properties, and also about probabilistic/stochastic systems, but we leave this out of our scope.

### 3 Effect of determinism

Determinism has been considered in P systems as early as 2003-2004, [31], [23]. In particular, it has been shown that deterministic P systems with symport/antiport are still computationally complete. Deterministic systems are interesting only as acceptors or as input to output processors, not generators.

Intuitively, determinism should require more cooperation, e.g., for appearance checking or zero test, instead of guessing and causing an infinite computation if the guess was wrong, fault-free decision must be used. Still, cooperative rewriting of degree two suffices, e.g., it is easy to simulate register machines.

The most famous and not-easy-to-see separation of computational power due to determinism is that catalytic P systems are computationally complete with two catalysts, [30], while deterministic catalytic P systems are not, [34] and [22]. It is via catalysts competing for objects that the objects exchange limited information crucial for meaningful computation.

The history and results of research in symport/antiport are presented in survey [29]. The size of a rule is the number of objects in its description. Rules of size two suffice for the computational completeness\(^3\), and in even with two membranes under certain conventions\(^4\). In the same time, deterministic symport/antiport systems are computationally complete with rules of size three, [23], but with rules of size two they are not, [44], (except in the tissue case, [17]). Indeed, deterministic systems with minimal communication cannot increase the total number of objects inside the system and halt.

---

3 Computational completeness is first shown already in [40] using both forms of minimal cooperation, in [20] for minimal cooperation by antiport, and in [14] for minimal cooperation by symport.

4 Either one superfluous symbol, [18], or the output in the skin, [16].
The best known results for \textit{evolution-communication P systems} (having non-cooperative evolution and symport/antiport rules of size at most two) are computational completeness with two membranes, \cite{1}, and with three membranes in the deterministic case, \cite{3}. Intuitively, deterministic zero-test checking has been implemented by moving a control symbol from region 2 into the elementary membrane or out to the skin.

Summarizing the above, determinism separates computational completeness from non-completeness for catalytic P systems and for P systems with minimally cooperative symport/antiport. The best known bounds on the number of membranes in computationally complete evolution-cooperation P systems are different in the deterministic and non-deterministic case. Still, one cannot underestimate the role of determinism for practical reasons like simplicity of analysis, and, more importantly, the result from one computation, not all of them.

One of the approaches to only use limited non-determinism is the so-called $k$-\textit{determinism}, \cite{36}, \cite{21}. In this way, we are only interested in one computation, while all deviations from it are easily detected. A recent continuation of this approach is to decrease or eliminate non-determinism by adding extra control to the derivation mode, calling it \textit{look-ahead mode}, \cite{42}.

We give another curious, but straightforward remark about \textit{minimal parallelism}. In systems that are deterministic in this mode, for every membrane there must always be at most one copy of rule applicable.

\subsection{Other models where non-determinism seems essential}

We mention other cases where the construction seem to rely on non-determinism in some important way.

The first case is P systems with active membranes, where except membrane separation, the rules are non-cooperative and the membrane structure is static. In \cite{15} and \cite{38}, authors give solutions to SAT problem. Naturally, different combinations of truth assignments are processed by different sub-membranes, each deciding whether the formula is satisfied, and sending some witness objects out to the skin membrane in the positive case. The problem was that only one answer object is needed. However, the skin membrane is not elementary, so it cannot be separated. Any other rule for witness objects is sequential, and it cannot be forced to stop until all witness objects undergo this rule, which might take non-polynomial number of steps. The authors invented the following (non-deterministic) solution: there is one target membrane in the skin, each of $s$ membranes with positive answer sends out two witness objects, that later enter either the target membrane or again membranes with positive answers. The “entering” of $2s$ objects in $s + 1$ membrane takes two steps, and the target membrane receives one or two objects. Minor additional work is done to make the system strongly confluent, i.e., all computations halt in the same configuration, but this is a less restrictive property than determinism.

The second case is non-cooperative P systems with promoters or inhibitors (of weight not restricted to one). Subtracting an element from a multiset is...
an essential detail of any non-trivial computation. However, if we have a supply of objects \( a \), even if controlled by higher-weight promoters or inhibitors, without non-determinism all of them will evolve in parallel, in the same way, so their number is either preserved or increased, or they are all erased. With non-determinism, such systems are shown to be computationally complete, \[19\]. Subtracting is done by non-deterministic rewriting of objects \( a \) by two rules. The system then checks if the second rule has been applied, and only once, by using promoters or inhibitors. Some additional work is done to ensure the ultimate confluence, which is less restrictive than confluence.

The third case is when a bound is imposed on the size of the alphabet, e.g., 5 or less, see the results in \[12\], \[11\]. Clearly, if we want to simulate some computationally complete device, e.g., a register machine, we typically need to represent its states by multiplicities. If we still have “enough” different symbols, states can be represented by incomparable multisets, e.g., \( \{ a^i b^j \mid i + j = s \} \). However, lowering the bound seems only possible if the states are represented by multisets like \( \{ a^i \mid i \leq s \} \), and ensuring that if \( a^j \) is used instead of \( a^i \), \( j < i \) (due to non-determinism), then the computation does not halt.

The fourth case is P systems without polarizations, where membranes can be divided and dissolved. It has still been shown to be computationally complete, \[4\], \[13\], by representing working registers of a register machine (or the sentential form of a matrix grammar with appearance checking) by multiplicities of membranes, incrementing by dividing a membrane and decrementing by dissolving one membrane (with an extra membrane of each type, because at least one is always needed to be divided). The zero-test is performed by keeping one membrane busy and testing whether any other membrane is available. This seems impossible to do deterministically, because there also exists an (unproductive, i.e., non-halting) computation, where keeping an extra membrane busy fails, and the single membrane is reported as available.

The fifth case is P systems with conditional uniport. Here, the interaction between objects is even weaker than in minimal symport/antiport; this is probably the most fragile model of P systems ever shown to be computationally complete, \[43\], \[27\]. All elementary operations seem impossible under determinism.

### 4 Effect of other variants of determinism, reversibility, confluence and similar properties

Reversibility in P systems has first been considered in \[37\], in an energy-based model. We start this section by summarizing some results on determinism, reversibility and their strong variants presented in \[8\], \[9\], \[7\] and \[10\]. Rewriting systems with full cooperation were considered, sometimes with (strong) priorities, promoters or inhibitors; we refer to these three features as “control”. Since membrane structure is fixed, the properties mentioned above are not affected if the structure is flattened, yielding one-region system. Notice that these systems are also equivalent to one-membrane symport/antiport systems. Moreover, the mentioned studies concern both maximally parallel and sequential systems.
Like determinism, reversibility is undecidable if either inhibitors or priorities are used (in maximally parallel case, determinism is undecidable even without control). However, strong determinism is a syntactic property (in sequential case, this property only yields one-rule systems). Strong reversibility is syntactic in the sequential case and decidable [33] in the maximally parallel case.

Table 1 summarizes the power of sequential (left) and maximally parallel (right) P systems, depending on the control features and underlying restrictions.

So what conclusions we can make from this? Determinism does not restrict the power of the systems, except in pure sequential case. Neither does reversibility, except maybe in pure cases. Strong determinism does in pure case, maybe also with inhibitors\(^6\), but not in other cases. Strong reversibility might decrease the power in all cases, but it has only been proved for pure systems.

We should mention the conjecture of non-universality of reversible P systems without control. Informally speaking, this is tied to the following facts: a) a universal computation seems to need zero-test or a similar form of appearance checking, b) the only known technique for that in systems without control is try-and-wait-and-then-check, c) if state change does not happen in the same step as consulting the symbol checked, then reversibility no longer holds.

The limitations of strong reversibility compared to (not-strong) reversibility are related to the fact that a similar simulation of a register machine would need to satisfy the uniqueness of previous configuration even in situations where multiple state symbols are present.

This, strong versions of determinism and reversibility may be desired due to their decidability and the motivation of reflecting certain microscopic physical properties. However, the design of such systems is more complicated, and sometimes impossible. Out of 14 cases of property-mode-control combinations, the separation is only proven for pure maximally parallel determinism VS strong determinism. Other 4 cases (pure sequential, priorities or priorities with inhibitor) show no limitation in the computing power, and 9 cases are unanswered.

---

\(^5\) Pri, pro, inh mean priorities, promoters and inhibitors, respectively.

\(^6\) One can generalize inhibitors to check for the absence of multiple higher-weight sets of objects, and the corresponding systems become at least as powerful as those with priorities, see details in [10].
Recall from the previous section that there exists models when even the (not-
strong) determinism is too restrictive. In Subsection 3.1, case 1 mentions strong
confluence, case 2 mentions ultimate confluence, and one may wish to attack the
latter three cases with, e.g., lookahead mode [42].

In general, deterministic VS confluent has been a very popular topic in paper
solving intractable problems by P systems in polynomial number of steps. This
has mainly been the framework of active membranes, and sometimes tissue sys-
tems with cell division or other models. In most cases confluent solutions have
been improved to deterministic ones (case 1 of Subsection 3.1 is an example of an
exception). For instance, [5] presents a solution where a lot of “extra” computa-
tion is only carried out so that the computation is deterministic (when multiple
objects can use the same resource in either order, the approach was to replicate
this resource enough times so that different copies are deterministically used in
parallel in one step). Notice also that different copies of the same object in the
same region are indistinguishable, so, e.g., \( a(ba) \Rightarrow \rightarrow ab \rightarrow ac \) and \( (ab)a \Rightarrow ab \rightarrow ca \) are not different. Likewise, two membranes with the same contents located in
the same region are indistinguishable, so, e.g., the steps below are the same.

\[
\begin{align*}
[a]_2 [b]_2 [c]_1 &\Rightarrow^a [b]_2 [c]_2 [a]_1, \\
[a]_2 [b]_2 [c]_1 &\Rightarrow^a [b]_2 [c]_2 [a]_1.
\end{align*}
\]

It deserves, however, some interest to investigate what penalty has determinism
in terms of descriptional or other complexity for different solutions of intractable
problems in polynomial time. For instance, generating truth assignments for
variables in an arbitrary order yields slightly simpler rules, but generating them
in the prescribed order makes it easier to signal the end of the generation phase.

5 Effect of other properties

We start with two properties relaxing the assumption of a global clock. The first
one (time-free) implies that the reactions that started at the same time will not
necessarily finish at the same time (see also Section 2), and then requires that
the results do not depend on the rule times. The second one (asynchronous)
implies that the reactions do not need to start immediately when their applica-
bility conditions are satisfied. Time-freeness has been proven undecidable, while
asynchronous systems are actually a derivation mode, but it can be viewed as a
syntactic restriction (with a modified halting condition).

Time-freeness has been introduced in [26], [25], with results improved in [6]
and then in [2]. For instance, it follows that evolution-communication P systems
are computationally universal with two membranes. However, consider a so-
called proton pumping restriction (where the only cooperative rules are moving
an object with a special object called a “proton”, and the proton does not appear
in other types of rules). Evolution-communication P systems are computationally
complete already with one proton, but the best known result computational
completeness for time-free systems is with four protons. These results can be
translated into the transitional P systems model: one bi-stable catalyst is enough for universality, while four bi-stable catalysts suffice in the time-free case.

Asynchronous P systems have been considered in [28], and some results and ideas belong to publications referenced there. We should also mention paper [35], relating this property to the corresponding knowledge in Petri nets. It is not difficult to see that for uncontrolled multiset rewriting, asynchronous systems lead to the same result as the sequential ones: their power is characterized by partially blind counter machines. Adding control, e.g., promoters, inhibitors or priorities, leads to universality, since it is easy to simulate of register machines where the computation, even under maximal parallelism, is, de facto, sequential.

6 Conclusions

The topic of this survey is restrictions of P systems induced by dynamic properties like determinism, strong determinism, reversibility, strong reversibility, confluence, strong confluence, always halting, time freeness, asynchrony. In particular, we try to answer the question what are limitations of such restricted classes of P systems compared to the unrestricted ones; sometimes a gap in computational power is known, sometimes it is conjectured, sometimes it is unknown, and and other times the power is the same. We note that, even in the latter case, there may be other penalties for having the property satisfied, such as descriptional complexity (number of membranes, weight of rules, the size of total description of the corresponding constructions).

For instance, determinism induces a gap in power for catalytic P systems, it requires different minimal weight of symport/antiport rules, whereas the best known number of bi-stable catalysts or number of membranes in evolution-communication P systems is different. Five other cases are mentioned, where one might expect the gap in power (membrane separation, non-cooperative controlled multiset processing, small alphabets, polarizationless active membranes, and conditional uniport); we give a number of informal comments to justify why.

Determinism and reversibility are compared with their strong versions and with unrestricted cases, depending on the controls used; open problems are also commented. The gap is only known in pure cases. Then determinism is compared to confluence. Finally, we speak about time freeness (the best known number of bi-stable catalysts is different) and asynchrony (with a gap for pure systems).

This survey does not pretend to completely cover all classes of P systems corresponding to the restrictions associated to the dynamic properties. One of its aims is to try to give a uniform perspective of the role of dynamic properties, and another aim is to encourage the membrane computing researchers to produce formal proofs of separation of computational power by the dynamic properties, e.g., in situations like those mentioned in Subsection 3.1.

Acknowledgements The author gratefully acknowledges the project RetroNet by the Lombardy Region of Italy under the ASTIL Program (regional decree 6119, 20100618).
References


45. P systems webpage. [http://ppage.psystems.eu](http://ppage.psystems.eu)
Bringing Biology and Engineering Together with Spatial Computing

Jacob Beal
BBN Technologies
10 Moulton Street Cambridge, MA, USA 02138
jakebeal@bbn.com

Abstract

Historically, natural biological organisms and engineered systems have had very different properties from one another: natural organisms tend to be robust and adaptable, while engineered systems tend to behave predictable and are simpler to modify. These two worlds are coming closer together, however, with our increasing understanding of biology and the ever-increasing complexity of engineered systems.

We have been transporting ideas from biology to engineering and vice versa, using a spatial computing approach, in which a collection of locally interacting devices are viewed as a discrete approximation of a continuous manifold. In this talk, I will describe how we build on this foundation to pursue applications in synthetic biology and morphogenetic engineering. In synthetic biology, we introduce compilers and constraint reasoning to transform spatial computing programs into DNA sequences controlling the behavior of living cells. In morphogenetic engineering, we augment the design of a robot with a developmental program for simulating its growth, which then guides variation of the design. Spatial computing thus serves as a seed for applying enabling biological principles to engineered systems and vice versa.
Abstract. In the past century, several conceptual and technological breakthroughs produced the digital computers and open the digital information age. At the very same time, the Watson–Crick model of the digital coding of the genetic information was developed. Despite this parallel development, biology as long focused in the understanding of existing systems shaped by natural evolution whilst computer science has built its own (hardware and software) objects from scratch.

This situation is no longer true: the emergence of synthetic biology opens the doors to the systematic design and construction of biological (fluid) machines. However, even if fluid machines can be based on a kind of digital information processing, they differ from the discrete dynamical systems we are used in computer science: they have a dynamical structure.

In this paper, we stress the parallel between the development of digital information processing and genetic information processing. We sketch some tools developed or appropriated in computer science that can be used to model and specify such fluid machines. We show through an example the use of MGS, a domain specific language, in the proof of concept of a “multicellular bacterium” designed at the 2007 iGEM competition.

Keywords: fluid machines, synthetic biology, computer modeling and simulation, (DS)²: dynamical systems with a dynamical structure, spatial computing, topological rewriting, domain specific language (DSL), MGS.

1 Introduction

In the preface of [10], Tibor Gánti highlights the divergence of information processing researches as they apply to biological systems (cells) or to artificial systems (computers). As early as 1944, Erwin Schrödinger speculates about “programs” and “genetic code” [34]. Since, information processing has been a constant source of fruitful analogies for genetics [25] and biology has provided many motivations and algorithms to computer science (from evolutionary algorithms to artificial neurons). But, despite the parallel developments of computer science and genetics, cf. Fig. 1, biology as focused in the understanding of existing
systems shaped by natural evolution whilst computer science has designed and studied its own hardware and software objects from scratch.

Considering biological entities, like cells or organisms, as “machine” has a long history. For instance, in the 17th century, when René Descartes tried to convince Queen Christina of Sweden that animals were just another form of machine, she is said to have replied: show me a clock that reproduces [1]. Three centuries were to pass before her question received an answer in 1951 with the publication of an article by John Von Neumann [40]. A machine, in the abstract and ideal form of a computation, could effectively build a copy of itself. Therefore, the reproduction argument cannot be used to distinguish in principle biological systems from machines.

The idea that living matter, and specifically cells, can be used as computers is appealing for several reasons: nanoscale devices, massive parallelism, low energy consumption, possible change in computability and complexity classes...and also the hope that computing with biological devices may give to the corresponding software properties usually attributed to living matter: autonomy, adaptability, self-repair, robustness, self-organization.

Tibor Gánti uses the term fluid machineries to describe machines based on chemical processes utilized in the living world. In [10] he introduces the chematon (chemical automaton), a minimal cell model composed of three stochiometrically coupled autocatalytic subsystems: a metabolism, a template replication process, and a membrane enclosing the other two. The qualifier “fluid” stresses the fact that, in contrary to electrical machines, real geometrical directions cannot be assigned to the energy exchanged between the components of the fluid automaton. Nevertheless, they can be described as dynamical systems with a (chemical) state that evolves in time. This is also the case for genetic process engineering [41,42], another example of chemical machines harnessing the cellular machinery. In this engineering discipline, existing genetic elements are modified to implement into cells biochemical logic circuits and programmed intercellular communication.
The objective is to achieve complex, predictable and reliable input/output cell behaviors.

However, we advocate that the term “fluid” also outlines another important property of living systems: they have a *dynamic structure*. Biological processes form highly structured and hierarchically organized dynamical systems, the spatial structure of which varies over time and must be calculated in conjunction with the state of the system (this is specially obvious in developmental biology). We call this type of system a *dynamical system with dynamical structure* [12,13], which we shall abbreviate to (DS)\(^2\).

The fact that the very structure of a biological system is dynamical\(^1\) has been highlighted by several authors; we can cite: the concept of *hyper-cycle* introduced by Manfred Eigen and Peter Schuster in the study of autocatalytic networks [6], the theory of *autopoietic systems* formulated by Humberto Maturana and Francisco Varela [39] or the concept of *biological organization* introduced by Walter Fontana and Leo Buss to formalize and study the emergence of self-maintaining functional structures in a set of chemical reactions [9]. The objective of all of these works has been to grasp and formalize the idea of change in the structure of a system, change that is coupled with the evolution of the state of the system.

Coming back to the question of harnessing biological processes to compute, it is interesting to follow the metaphor: if we want to use cells as computing devices, what makes a population of idealized cell intrinsically different from a Turing machine? It may be that, from a technical point of view, there is no difference, meaning that Turing computation and “cell computation” coincide in term of computability. However, the computing devices differ definitively, in the same way that lambda expressions differs from Turing machines.

A Turing machine has a fixed structure. The tape is unbounded and only a finite part of the tape is used during the computation; however, the structure of the tape is fixed *a priori*: a sequence of symbols. This means that the control of the machine is also fixed: the head can move only to the left or to the right. The writing of a symbol on the tape is fully determined by the state of the control automaton of the Turing machine and a symbol in its (left or right) neighborhood. We can say that the control part of the Turing machine (control automaton and neighborhood) is predefined. At first sight, the situation does not seem too much different for a population of cells. Obviously the cells in a tissue share the same genetic program and two cells interact and change accordingly their state\(^2\) because they are neighbors. And the dynamic organization of a set of cells can be coded in someway into a linear tape. However this coding is not straightforward at all. The structure of a living system is not predefined: cells growth, divide and their emerging organization as tissue or organisms exhibits a great variety that must be computed with the system evolution and cannot be predicted by the

---

\(^1\) Biological systems are not the only ones that may exhibit a dynamic structure. For example, in control theory, these questions have also been addressed for instance with *systems of variable structure* [38].

\(^2\) The state of a cell is a complex value that changes due to internal processes. But the state of a cell can also change because of the interactions with the cells in the neighborhood through diffusion and active transport, by endo- and exocytosis, by mechano-transduction, by propagation of an electric field, etc.
simple inspection of the genetic program. Moreover, this neighborhood is dynamic. It means that the control part of a biological system (genetic program and neighborhood) cannot be fixed a priori.

The modeling, “the programming”, the simulation and the analysis of such (DS)² raise a difficult problem: how to define an evolution function when its set of arguments (the state variables) is not completely known at the specification time?

In the rest of this paper we sketch a formalism and an experimental programming language called MGS, which addresses this problem. MGS relies on a notion of “spatial structures rewriting” to specify local interactions between the system components. This approach enables the specification of the components evolution as well as their dynamic organization.

2 A Brief Introduction to MGS

2.1 Lessons from L Systems and P Systems

Computer Science has developed many languages and tools to help model and simulate dynamical systems. P systems (membrane computing) [30] and L systems [33] are examples of bio-inspired formalisms that have been successfully used in the modeling of biological (DS)². We will motivate MGS concepts based on relevant features of these two formalisms.

A Common Computational Mechanism. First, one can note that P and L systems share the following three characteristics.

Discrete Space and Time. The structure of the state (the membranes hierarchy in a P system, the parenthesized string in a L system) consists of a discrete collection of values. This discrete collection of values evolves by discrete steps. We call “spatial” the organization of the elements in the collection because this structure does not unfold in time.

Temporally Local Transformation. The computation of a new value in the new state depends only on values for a fixed number of preceding steps (and as a matter of fact, only one step).

Spatially Local Transformation. The computation of a new collection is done by a “structural combination” of the results of more elementary computations involving only a “small and static subset” of the initial collection. “Structural combination”, means that the elementary results are combined into a new collection, irrespectively of their precise value. “Small and static subset” makes explicit that only a fixed subset of the initial elements are used to compute a new element value (this is measured for instance by the diameter of an evolution rule in a P systems, or the context of a rule in a L system).

A Rewriting Mechanism. Considering these shared characteristics, the abstract computational mechanism is always the same: (1) a subcollection A is selected in a collection C; (2) a new subcollection B is computed from the collection A; (3)
the collection $B$ is substituted for $A$ in $C$. These three basic steps are reminiscent of the notion of rewriting and, indeed, P systems can be described as multiset rewriting and L systems as a kind of string rewriting. However, we prefer to call it a transformation because the notion of rewriting is mainly developed on terms (strings are terms in an associative monoid and multisets are terms in an associative and commutative monoid) and we want to be more general.

In addition to transformation specification, there is a need to account for the various constraints in the selection of the subcollection $A$ (the pattern language) and the replacement $B$, and in the rule application strategy. For example, for L systems and P systems, the basic rule application strategy is the maximal parallel one.

**Locality and Interaction.** From the perspective of the simulation of $(DS)^2$, several features are appealing in the L systems and in the P systems formalisms.

First, summing up the local evolutions triggered by the rules specifies the global evolution of the state. So there is no need to have a global picture of the state to specify its evolution.

Secondly, elements referred in a rule are referred implicitly through pattern matching. For instance, a pattern $a$ refers to some occurrence of $a$ in the collection: there is no need to use a global reference to $a$. A global reference (like a coordinate) may be difficult to maintain as the collection evolves (for instance when new symbols are inserted elsewhere).

Third, element in a pattern are related through the relationships induced by the organization of the collection. For example, in a string, the pattern $ab$ denote two symbols $a$ and $b$ that must be consecutive in the string. For multisets, each element is neighbor to all the other elements of the multiset.

### 2.2 The Topological Organization of the State

After this presentation, the main difference between the two formalisms, for the purpose of simulation, appears to be the organization of the collection: an imbrication of multisets of symbols or a parenthesized string of symbols. Thus, our idea is to generalize the approach of P and L systems by developing a framework where multiple organizations can be considered uniformly. This is the notion of topological collections introduced in [19] to describe arbitrary complex spatial structures that appear in biological systems [15] and other dynamical systems with a time varying structure [16,20].

The definition of topological collection is based on a mathematical device developed in algebraic topology: the notion of chain [23] which extends the notion of labeled graph.

**Incidence Structures.** An abstract combinatorial complex $K = (C, \prec, \dim)$ is a set $C$ of abstract elements, called topological cells, provided with a partial order $\prec$, called the boundary relation, and with a dimension function $\dim : C \to \mathbb{N}$ such that for each $c$ and $c'$ in $C$, $c \prec c' \Rightarrow \dim(c) < \dim(c')$. The reader must pay attention not to confuse biological and topological cells. We write $c \in K$ when a cell $c$ is a cell of $C$. A cell of dimension $p$ is called a $p$-cell. We say that a cell $c'$
is a border of a cell $c$ if $c' \prec c$. The boundary of a cell $c$ is the set of borders of $c$. The faces of a $p$-cell $c$ are the $(p-1)$-cells $c'$ such that $c' \prec c$ and we write $c > c'$ or $c' < c$; $c'$ is called a coface of $c$. Two cells $c$ and $c'$ are $q$-neighbors either if they share a common border of dimension $q$ or if they are in the boundary of a $q$-cell (of higher dimension).

A cell of dimension 0 corresponds to a point, a 1-dimensional cell corresponds to a line (an edge), a cell of dimension 2 is a surface (e.g., a polygon), etc. For example, a graph is an abstract combinatorial complex (ACC) built only with 0- and 1-cells. Another example is pictured in Fig. 2.

**Topological Collections.** The next step is to attach a value to the cells of a complex. Algebraic topology takes this value in a commutative group since it gives a natural group structure to the set of chains [23]. We relax this assumption for topological collections: a topological collection $C$ is a function that associates a value from an arbitrary set $V$ with cells in an ACC, see Fig. 2. Thus the notation $C(c)$ refers to the value of $C$ at cell $c$; $C(c)$ is called the label of the cell $c$. Labels can be used to capture geometric properties or to represent the arbitrary state of a subsystem (a mass, a concentration of chemicals, or a force acting on certain cells).

We write $|C|$ for the set of cells for which $C$ is defined. The collection $C$ can be written as a formal sum $\sum_{c \in |C|} v_c \cdot c$ where $v_c \overset{df}{=} C(c)$. With this notation, the underlying ACC is left implicit but can usually be recovered from the context. By convention, when we write a collection $C$ as a sum

$$C = v_1 \cdot c_1 + \cdots + v_p \cdot c_p$$

we insist that all $c_i$ are distinct. This notation is directly used in MGS to build new topological collections on arbitrary ACC of any dimension. Notice that this addition is associative and commutative: the order of operations used to build a topological collection is irrelevant.

In the MGS programming language, topological collections correspond to aggregate data types. These data types differ by the specification of their underlying

![Fig. 2.](image)

On the left, the Hasse diagram of the boundary relationship of the ACC given in the middle: it is composed of three 0-cells ($c_1, c_2, c_3$), of three 1-cells ($e_1, e_2, e_3$) and of a single 2-cells ($f$). The three edges are the faces of $f$, and therefore $f$ is a common coface of $e_1$, $e_2$ and $e_3$. On the right, a topological collection associates data with the cells: positions with vertexes, lengths with edges and area with $f$. 
cellular complex. In the current implementation of the MGS language, usual data structures (records, sets, sequences, trees, arrays, etc.) are represented by special kinds of one-dimensional topological collection, namely vertex-labeled graphs: elements of the data structure are attached to the vertexes and the edges represent the relative accessibility from one element to another in the data structure. MGS also handles more sophisticated spatial structures corresponding to arbitrary ACC of any dimension.

2.3 Topological Rewriting

The next move is to define a suitable notion of topological collection transformation. As mentioned in the introduction, the transformation of a topological collection must be able to express changes in the labels as well as changes in the underlying spatial structure.

A dedicated definition has been developed in [19]. However, thanks to the term structure of a topological collection, transformations can be defined in the framework of set rewriting, following an approach similar to that taken in [31] for hyper-graphs: using the additive representation of topological collections, topological rewriting can be simply defined as an adapted version of conditional first-order associative-commutative term rewriting, see [36] for the details.

A transformation $T$ is a function specified by a set of rewriting rules \{ $p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n$ \} where each $p_i$ is a pattern and each $e_i$ is an expression. An application of a rule matches a sub-collection with $p_k$ that is then substituted by the result of expression $e_k$. In rewriting rules, patterns match sub-expressions, that is, partial sums of the whole sum representing the topological collection that the rule is applied on. It is in this sense that the additive structure of topological collections is preserved (but a transformation is not necessarily an homomorphism).

**Patterns.** The formal definition of topological rewriting is less interesting than the actual syntax of the pattern language used to specify the left hand side (lhs) of a rewriting rule: as a matter of fact, the lhs of a rule must match a sub-collection, that is a subset of $\mathbb{C}$ and a sub-relation of the incidence relation $\prec$ of the complex $K$. This information can be difficult to specify without the help of a dedicated language. We have studied several pattern languages. We use here a very small fragment of the MGS path pattern language. Path patterns can be composed with respect to the neighborhood relationships but we don’t use this feature in the example developed in this paper.

**Pattern Variables.** A pattern variable $x$ matches a cell and its label. Patterns are linear: two distinct pattern variables always refer to two distinct cells. The identifier $x$ can be used elsewhere in the rule to refer to the label of the matched cell; the cell itself can be referred through the special identifier $\hat{x}$. This convention avoids the introduction of two identifiers to match a cell and its associated value. Using the additive notation for topological collections, and without the previous convention, this pattern is translated to $x \cdot \hat{x}$ where the variable $x$ ranges over the labels, and where the variable $\hat{x}$ ranges over the cells.
Conditional rules. A guard can be used to specify a condition that must be satisfied by the matching. For instance, expression $x/x > 5$ matches a cell $x$ labeled by an integer $x$ greater than 5.

Strategies. Rule applications are controlled through a rule application strategy. Several strategies are available in MGS like the maximal parallel application and the Gillespie stochastic simulation algorithm used in the simulation of chemical reactions [35]. These strategies control the advancement of time in the simulation (synchronous, asynchronous, stochastic, etc.). They are often non-deterministic, i.e., applied on a collection $C$, only one of the possible outcomes (randomly chosen) is returned by the transformation.

Multisets as “Free” Collections. Let an ACC $K = \{\bot, c_1, c_2, \ldots\}$ where the $c_i$ are incomparable and $\bot < c_i$. The corresponding topological collection is a multiset and the associated notion of transformation corresponds to classical multiset rewriting. In this sense, any topological collection can be obtained from a multiset by putting additional constraints on $\prec$.

Note that the previous definitions of topological collections and transformations are useful for developing a unified simulation framework but have less interest if one is concerned by the derivation of properties specific to an underlying topology.

3 The modeling of a “multicellular bacteria”

In this section, we illustrate the use of the MGS framework for the modeling and the simulation of a fluid machine.

3.1 The international Genetically Engineered Machine (iGEM) Competition

The emergence of synthetic biology [8,22,4,24] opens the doors to the systematic design and construction of biological (fluid) machines. After the construction of the first artificial genetic regulatory networks in E.coli around 2000 [11,7], this domain mainly develops around the engineering of synthetic gene network [21,27,5]. It has been largely popularized through the iGEM competition, a yearly competition launched by the MIT in 2003 [3]. The competition is aimed at undergraduate students that are given the opportunity to design, model and assemble BioBricks [26] to produce new biological functions integrated into living systems. More than 160 teams coming from all around the world participate in the 2011 issue.

3.2 Objectives of the Synthetic Multicellular Bacterium Project

The 2007 French team supervised by Ariel Lindner and Samuel Bottani participated in the competition and was ranked first in the “foundational research” category for their Synthetic Multicellular Bacterium project. Unlike most projects
involving a regulatory network functioning in a single cell following a straight-forward sensing/transduction/actuation loop, the functioning of the Synthetic Multicellular Bacterium is implemented at the population level. MGS was used to produce most of the simulations needed to validate the design (one simulation was done in MATLAB). To save room, we present here only the design idea of the Synthetic Multicellular Bacterium and one of the supporting MGS models. The interested reader may found additional information in [37] where several simulations that are inspired or that extend the initial Synthetic Multicellular Bacterium simulations are presented.

The project is aimed at the design of a synthetic multicellular bacterium. This organism would allow the expression of a lethal or dangerous transgenic gene in the *Escherichia coli* bacterium without disturbing the development of its biomass. The main difficulty was to install a mechanism of irreversible bacterial differentiation that makes possible to express the transgene only in a part of the population unable to reproduce. The two lines, germinal (not differentiated) and somatic (differentiated and unable to reproduce), are interdependent and then constitute a multicellular organization (hence the name “*multicellular bacterium*”). To ensure that the ratio between the two populations makes it possible for the system to grow, the sterile somatic cells are designed to provide to the germinal cells a molecule essential to their reproduction: DAP (diaminopimelate). Fig. 3 sketches the general principle of the project. This design asked for the development of two distinct biological functionalities, one for the cellular differentiation and the other for the feeding of DAP to the germinal cells.

This design is an example of a fluid machine: it is the dynamic organization of the whole population (into two differentiated subpopulations) that is viable and functional. A single cell cannot survive: it is either sterile or unable to reproduce alone. The Paris team provided experimental evidences and theoretical proofs that the Synthetic Multicellular Bacterium organism is viable.

---

**Fig. 3.** The Synthetic Multicellular Bacterium is composed of two cell types: germ cells (G) and somatic (S) cells. G cells are able to live by producing two different types of cells: G cells and S cells. S cells are derived from G cells by an irreversible differentiation step, exhibiting a new function required for the survival of the G cells. S cells cannot reproduce. This dependency between G and S cells defines the organism. Additional informations is available at [2,37].
3.3 The Paris Team Proposal

The Paris team has proposed an original construction to implement this functionality into the E.coli bacterium. The gene regulatory networks of the proposal is described in Fig. 4. Two functions are described: a feeding device based on the production of DAP molecules (light gray) and a differentiation device based on a classical Cre/LOX recombination scheme (dotted box).

In the germline G, the expression of ftsK, a gene essential for replication, is controlled by its natural promoter. On the contrary, the dapA gene is not active since it lacks a promoter to initiate its transcription and G is auxotrophic in DAP (the corresponding protein product of the gene dapA). DAP diffuses in the environment and is rapidly degraded.

A dapAp promoter is sensitive to DAP concentration and it is located before the gene Cre. The production of Cre, in presence of DAP in the environment, initiate the recombination/differentiation process.

After recombination, the genomic reassembly leads, by the excision of the parts between the two LOX genes, to the soma cell type S (and a plasmid that is rapidly degraded). In the feeding device S, DAP is under the control of its constitutive promoter and can be expressed. The synthesized DAP diffuses in the environment allowing to reach G cells. Lacking ftsK genes, S cells are sterile and eventually die.

3.4 One MGS model of the Synthetic Multicellular Bacterium

Several models of the Synthetic Multicellular Bacterium have been developed to study, through simulation, various questions. We present here a model that integrates a simple mechanical and a biological behavior.
In this example, we abstract individual biological cells by disks localized in a 2D Euclidean space (the third dimension is not considered as the Synthetic Multicellular Bacterium is supposed to grow in the plane of a petri dish). Cells push away each other and consequently change their positions in space and their immediate neighborhood. Thus, this neighborhood is required to be dynamically computed according to the position of the disks in the plane. The state of a cell is described by a record:

```
record bact = { x, y, vx, vy, fx, fy, size, radius, dap, soma }
```

which includes the position, velocity and force exercised on the cell, its radius, the local DAP concentration, and the differentiation state (germinal or somatic).

Our approach is based on the specification of cell-cell dynamical interactions and the computation of the neighborhood of the cells using an implicit Delaunay triangulation. This approach has already been used in systems biology for the modeling of cell population [32]. The MGS declaration

```
delaunay D(c:bact) = [c.x, c.y]
```

defines Delaunay collection called D: a graph where nodes are cells and edges are computed implicitly by the run-time using the vector of coordinates extracted from the cells using the function D.

**Description of the Model.** The modeling of Synthetic Multicellular Bacterium is organized into two coupled models: a mechanical model and a biological model.

**The Mechanical Model.** The mechanical model consists of a mass/spring system. Bacteria are considered as punctual masses localized at the center of their associated circle; the presence of a spring between two masses depends on the neighborhood computed by the Delaunay triangulation. The mechanical effect of the growth of the bacteria is captured by the elongation of the springs rest lengths. Each cell computes its acceleration by summing all mechanical forces induced by its incident springs, and consequently moves in space. This is done by the transformation Meca. Meca sums the forces applied on each cell using a `neighborsfold` expression. A naive Euler schema is used twice to integrate during the time step, acceleration into velocity and velocity into new positions.

```plaintext
fun interaction(ref, src) =
  let x = ref.x - src.x and y = ref.y - src.y in
  let dist = sqrt(x*x+y*y) in
  let spring = 0.0-K*(dist-(ref.radius+src.radius))/dist
  in fx=x*spring - ref.vx*MU, fy = y*spring - ref.vy*MU

fun addVect(u, v) = fx = u.fx + v.fx, fy = u.fy + v.fy
fun sum(x, u, acc) = addVect(acc, interaction(x,u))

trans Meca[Dt] = {
```

```plaintext
  fun interaction(ref, src) =
    let x = ref.x - src.x and y = ref.y - src.y in
    let dist = sqrt(x*x+y*y) in
    let spring = 0.0-K*(dist-(ref.radius+src.radius))/dist
    in fx=x*spring - ref.vx*MU, fy = y*spring - ref.vy*MU
  fun addVect(u, v) = fx = u.fx + v.fx, fy = u.fy + v.fy
  fun sum(x, u, acc) = addVect(acc, interaction(x,u))
```
The overall form of the unique rule of the Meca transformation is $e \Rightarrow v \cdot \hat{e}$ which means that this rule only update the value associated to the cell matched by $e$.

The computation of the value $v$ requires some explanation. Variables in capital represent parameters of the model (constant defined elsewhere). The definitions interaction, addVect and sum specify auxiliary functions. All functions are curried in MGS: so, sum(e) is a function awaiting the remaining arguments u and acc. The $+$ operator between records denotes the asymmetric merge. The expression $r_1 + r_2$ computes a new record $r$ having the fields of both $r_1$ and $r_2$: $r.a$ has the value of $r_2.a$ if the field $a$ is present in $r_2$, otherwise it has the value of $r_1.a$. The expression neighborsfold($f$, init, $c$) iterates a binary reduction function $f$ over the labels of the neighbors of $c$ to build up a return value. The argument init is used to initialize the accumulator. Note that the neighborsfold operator rely on the ACC structure underlying the topological collection to determine the neighbors of $c$.

The Biological Model. The DAP diffusion dap_diff is modeled by a classical continuous model (numerical integration of the diffusion equation is done using a simple Euler explicit schema). New rules are added to deal with cellular growth, division and death: in presence of DAP, G cells grow by increasing their radius. When the G cell radius reaches a threshold, the cell divides. S cells keep on growing then die when another threshold is reached. The corresponding transformation is called Cell and computes the evolution during a time step $Dt$:

```plaintext
fun divide(b) = 
  (b + { size = (b.size / 2.0), ...}) * newcell(D),
  (b + { ...}, x = noise(b.x), y = noise(b.y)) * newcell(D)

trans Cell[Dt] = {
  x / x.soma & x.size > 4 => if random(1.0) < 0.01 then <undef> else x * \x
  x / x.soma => (x + { size = x.size + Dt*GRate }) * \x
  x / x.size < 2 => let dap_diff =
    neighborsfold(\y,acc.acc + Dt*DIFF*(y.dap-x.dap), 0, x) in
    let dap = dap_diff.dap / neighborcount(x) - Dt*CONS
    in if (dap <= Mdap) then
      if (random(1.0) < DProb)
        then (x + { soma = true }) * \x
      else (x + { dap = dap }) * \x
    else (x + { dap = x.dap + dap, size = ... }) * \x
```
The Modeling and the Simulation of the Fluid Machines...

The first rule gives the fate of a somatic cell of size greater than 4: it goes to apoptosis (cell death) with a probability of 1% per $Dt$ period. The second rule gives the fate of a somatic cell with a size less or equal to 4 (rules are tried in the order of their specification). Such cells increase in size. Third rule apply to germinal cell of size less than 2. Such cells acquire DAP from the environment. If the amount of DAP in the environment is below a given threshold $Mdap$, the cell can spontaneously turn to a somatic cell, with some probability.

The fourth rule is interesting: it specifies the division of a bacterium if some conditions are meet. The function \texttt{divide} returns two new labeled cells (new cells for collection of type $D$ are build by the expression \texttt{newcell(D)}). The first has the same coordinates as the argument. The second cell has the same coordinates perturbed by some small noise. The mechanical evolution will separate quickly the two cells and will reorganize the whole structure. This reorganization will impact the diffusion of DAP. So there is a feedback loop between the spatial organization of the system and the process inhabiting this organization.

\textbf{Integration of the Two Models.} As classical functions, transformations can be arbitrarily composed. This is the key to the coupling of the two models. The iteration of a function can be specified by the MGS option \texttt{iter}. It allows to deal with different time scales: assuming that the mechanical process is faster than the cellular process the whole model is captured by the following evolution function:

\begin{verbatim}
fun SMB(state) = Cell[Dt=\Delta_2 t](Meca[Dt=\Delta_1 t, iter=\Delta_2 t/\Delta_1 t](state))
\end{verbatim}

where the named argument $Dt$ corresponds to the time step used in transformations \texttt{Meca} and \texttt{Cell}. Here transformation \texttt{Meca} is applied $\Delta_2 t/\Delta_1 t$ times for only one application of \texttt{Cell}.

\section{Conclusions}

The example above remains simple: there is no need for sophisticated pattern-matching, yet it is a (DS)$^2$ and it exhibits clearly a feedback loop between the spatial structure and the processes inhabiting the structure. The conclusions that are drawn below are also supported by other applications.

The MGS description is concise thanks to the notion of collection and the notion of transformation, which unify (for the purpose of the simulation) the handling of a wide family of dedicated data structures [14]. The MGS specification follows the natural structure of the model: there is generally one evolution rule for each type of evolution. Evolution rules can express simultaneously structural as well as state changes. They can be grouped into transformation associated to
some kind of physical laws (mechanics, chemistry, etc.) making manifest multiphysics simulation (i.e., involving multiple physical domains). The coupling between transformations is easily controlled because the functional nature of a transformation. This enables multiscale models. Although transformations are associated to discrete step evolution, the example shows that numerical integration of continuous processes can be integrated smoothly with discrete evolutions (such as cell division in the rule 4 of the transformation Cell). The alternative models developed for the Synthetic Multicellular Bacterium (each model focuses on a specific time scale using a dedicated theoretical framework) outline the versatility of the approach. For instance, stochastic model can be expressed simply through the choice of a dedicated rule application strategy [35].

Past work on MGS have focused on the development of a framework relevant for simulation and the validation of MGS concepts through numerous examples, including application in system and synthetic biology; see the MGS home page: http://mgs.spatial-computing.org. The theoretical investigation of the topological framework (e.g., can we develop a natural complexity notion on patterns and on topological operations? Can we develop a dedicated static analysis framework for MGS programs? Is there a relevant notion of model-checking? Etc.) is underway.

Acknowledgements

First thanks must go to Antoine Spicher and Olivier Michel for our long standing collaboration within the MGS project and to the 2007 iGEM Paris team. Thanks also go to Hanna Klaudel and Franck Delaplace at IBISC and the members of the SynBioTIC project for their advices and fruitful discussions. This presentation is based on work partially presented in [18,19,12,28,17,37,29]. Parts of this work have been funded by Genopole, University of Evry and CNRS, IBISC and IRCAM labs, and by the AutoChem and SynBioTIC ANR projects.

References


33. Ronzenberg, G., Salomaa, A. (eds.): L systems: from formalism to programming languages. Springer Verlag (Feb 1992)


Parallel and Distributed Algorithms in P Systems
(Brief outline)

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

Abstract. This talk discusses our group’s recent use of P systems to model parallel and distributed algorithms. Several framework extensions are motivated and analysed, in particular, reified cell IDs, complex symbols, generic rules, asynchronous operational modes, asynchronous complexity and related properties, such as safety and liveness. We illustrate our proposals via P system models of several well-known distributed algorithms, such as leader election and minimum spanning trees and, as another type of application, a dynamic programming algorithm for stereo matching in image processing. We suggest criteria to assess the merits of this approach and offer preliminary evaluations of our proposed additional ingredients, which have proved useful in refactoring existing systems and could be useful to the larger P system community.

Keywords: P systems, cell IDs, complex symbols, generic rules, distributed algorithms, parallel algorithms, synchronous networks, asynchronous networks, leader election, minimum spanning trees, stereo matching.

1 Introduction

A P system is a parallel and distributed computational model, inspired by the structure and interactions of cell membranes. This model was introduced by Păun in 1998–2000 [28]. An in-depth overview of this model can be found in Păun et al. [29]. This talk discusses our group’s recent use of P systems to model parallel and distributed algorithms, direction pioneered by Ciobanu et al. [8, 9]. We intentionally use a maximalist approach, by selecting only the most adequate ingredients (but no more) and propose several extensions, which seem useful or even required for distributed or otherwise complex algorithms.

2 Preliminaries—Basic Model

While we use our own version of P systems, our core results should remain valid and meaningful for other versions of these systems. As seen in most of our recent
papers \[1, 10, 12, 13, 11, 14–16, 21, 24–26\], our preferred membrane structure is a directed graph (digraph) or one of its subclasses, such as a directed acyclic graph (DAG) or, occasionally, a tree; (undirected) graph structures can be emulated by symmetric digraphs. Arcs represent duplex channels, so parents can send messages to children and children to parents (i.e. all communications follow the direct or reverse direction of digraph arcs).

Each arc has two labels—at its tail and at its head. Arc labels can be used for directing messages to a specific target cell. An arc, \(\alpha = (\sigma_i, \sigma_j)\), which is not explicitly labelled, is implicitly labelled with the indices of the two cells, i.e. in this case, \(\alpha\)'s labels are: \(j\)—at tail, close to \(\sigma_i\); \(i\)—at head, close to \(\sigma_j\).

Figure 1 (a) shows an arc from \(\sigma_1\) to \(\sigma_2\), implicitly labelled by cell indices: \(2\)—at tail; \(1\)—at head.

In the basic model, all cells evolve synchronously. Rules are prioritized and applied in weak priority order. The general form of a rule \([15, 26]\), which transforms state \(S\) to state \(S'\), is

\[
S \xrightarrow{\alpha} S' \ x' \ (y) \beta \ldots |z,
\]

where:

- \(S, S'\) are states, \(S, S' \in Q_i\);
- \(x, x', y, z\) are strings which represent multisets of symbols, \(x, x', y, z \in O^*\);
- \(\alpha\) is a rewriting operator, \(\alpha \in \{\text{min}, \text{max}\}\);
- \(\beta\) is a transfer operator, \(\beta \in \{\uparrow, \downarrow, \leftrightarrow | \gamma \in \{\forall, \exists\} \cup A\}\), where \(A\) is the set of (implicit or explicit) arc labels.

The transfer operator \(\beta\)'s arrow points in the direction of transfer: \(\uparrow\)—towards parents; \(\downarrow\)—towards children; \(\leftrightarrow\)—in both directions. Note that, if all rules exclusively use \(\leftrightarrow\) arrows, then arcs’ directions do not matter: this offers an alternate way to consider graphs, at the rule level (the other way, at the structural level, is to consider symmetric digraphs).

The transfer operator \(\beta\)'s qualifier, \(\gamma\), indicates the distribution form: \(\forall\)—a broadcast; \(\exists\)—an anycast; an arc label—a unicast over a specific arc (i.e. to a specific target).

Although the definition does not enforce this, we typically ask \([24]\) that all cells start with identical state and rule sets (as being mass-produced by a virtual cell factory); cells should only differ in their initial contents and their relative position in the structural digraph. This is a strong requirement; this precludes custom rule sets for each cell, but, as we will see later, enables the design of complex algorithms with fixed size state and rule sets, which do not change, regardless how many cells are included in the system.

The above definition corresponds to what we called a simple P module \([11]\). Figuratively, if we allow arc “stumps”, i.e. disconnected arc heads and tails, we obtain more general P modules \([12]\). P modules define a controlled way for recursive composition of P systems; however, this is another complex topic, which we do not pursue here.
3 Extensions—Leader Election (Sync)

Let us use the well-known distributed leader election problem and its famous impossibility result, as presented by Lynch [23] and Tel [30], to highlight the need for reified cell IDs and discuss a simple solution. Leader election is a fundamental problem in distributed algorithms and can be viewed as a highly abstract and simplified version of the cellular differentiation in developmental biology. Imagine a network of cells which must elect a leader. A celebrated result shows that this is impossible, if the system is totally symmetric, e.g., if the network is a circular ring and all cells are totally identical. This might be our case, if all our cells start with the same rule set, same state and same contents. While we have decided to keep the same rule set for all cells, we can still solve the problem by letting each cell, $\sigma_i$, start with its own unique cell ID symbol, $\iota_i$. We thus reify the external cell index $i$ into an internal symbol, which is accessible to the rules; in fact, we will use it exclusively as an immutable promoter [26].

Figure 1 (b) shows a ring structured system, where cells contain cell ID symbols, $\iota_i$; this breaks the symmetry and enables the leader election process.

![Diagram](image)

**Fig. 1.** (a) An implicitly labelled arc. (b) A ring structured system, where cells contains cell ID symbols.

Also, we enhance our vocabulary by recursive composition of elementary symbols from $O$ into a simple form of complex symbols [26]. Such complex symbols can be viewed as complex molecules, consisting of elementary atoms or other molecules. Further, complex symbols let us process our multisets with high-level generic rules, using free variable matching. We argue the practical necessity of this approach:

1. it enables reasonably fast parsing and processing of subcomponents (practically impossible with string objects);
2. it allows us to describe an algorithm with a fixed size elementary alphabet and a fixed sized rule set, neither of which depend on the number of cells in the system (sometimes impossible with only atomic objects).

To explain these additional ingredients, consider this rule:
This is a generic rule, which uses an extended rewriting mode, with complex symbols, $c_i$ and $n_j$, where $i$ and $j$ are free variables. In fact, $c_i$ and $n_j$ are just shorthands for tuples $(c, i)$ and $(n, j)$, or, equivalently, for compound terms $c(i)$ and $n(j)$. If needed, we can build more complex symbols by recursive composition; e.g., we could have complex symbols such as $d(e, i, f(j))$. Generally, a free variable could match anything, including another complex symbol. However, in this rule, $i$ and $j$ are constrained to match cell ID indices only:

1. $i$—because it also appears as the cell ID of the current cell, $\iota_i$;
2. $j$—because it also indicates the target of the transfer mode, $\iota_j$, if we assume that arcs are implicitly labelled by cell indices.

A generic rule is identified by using an extended version of the rewriting mode, here $\min \min$. Briefly:

1. according to the first $\min$, this rule is instantiated once, for one of the existing $n_j$ symbols (if any), while promoter, $\iota_i$, constrains $i$ to the cell ID index of the current cell, $\sigma_i$;
2. according to the second $\min$, the instantiated rule is applicable once, i.e. if applied, it consumes one $a$ and one $n_j$, produces one $b$ and sends one $c_i$ to neighbor $j$ (if this neighbor exists, as parent or child).

In fact, we consider all four possible combinations of the instantiation and rewriting modes: $\min \min$, $\min \max$, $\max \min$, $\max \max$. The interpretation of $\min \min$, $\min \max$ and $\max \max$ modes is straightforward. While other interpretations could be considered, the mode $\max \min$ 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. The instantiations are ephemeral, created when rules are tested for applicability and disappearing at the end of the step.

As an example, consider a system with $N$ cells, $\sigma_1, \sigma_2, \ldots, \sigma_N$, where cell $\sigma_1$ has two structural neighbors, $\sigma_2$ and $\sigma_3$, is in state $S_3$ and contains multiset $a^2 n_j^3 n_3$. Consider also all possible instantiations of the following rule, $\rho_a$, where $a$ is one of the four extended rewriting modes:

\[
(\rho_a) \ S_3 \ a \ n_j \to a \ S_4 \ b \ (c_i)_{\iota_j} \ | \iota_i.
\]

- Rule $\rho_{\min \min}$ generates one of the two low-level instances:
  either $S_3 \ a \ n_2 \to_{\min} S_4 \ b \ (c_1)_{\iota_2}$ or $S_3 \ a \ n_3 \to_{\min} S_4 \ b \ (c_1)_{\iota_3}$.
- Rule $\rho_{\min \max}$ generates one of the two low-level instances:
  either $S_3 \ a \ n_2 \to_{\max} S_4 \ b \ (c_1)_{\iota_2}$ or $S_3 \ a \ n_3 \to_{\max} S_4 \ b \ (c_1)_{\iota_3}$.
- Rule $\rho_{\max \min}$ generates the two low-level instances:
  $S_3 \ a \ n_2 \to_{\min} S_4 \ b \ (c_1)_{\iota_2}$ and $S_3 \ a \ n_3 \to_{\min} S_4 \ b \ (c_1)_{\iota_3}$.
- Rule $\rho_{\max \max}$ generates the two low-level instances:
  $S_3 \ a \ n_2 \to_{\max} S_4 \ b \ (c_1)_{\iota_2}$ and $S_3 \ a \ n_3 \to_{\max} S_4 \ b \ (c_1)_{\iota_3}$.
These generated instances are then considered for application, as in the basic model. Without our new ingredients, each cell would need a much larger logically equivalent custom rule set. For example, instead of $\rho_{\text{max}}$, cell $\sigma_1$ would need its own custom rule set, of $N$ low-level rules:

$$\{ S_3 \ a \ n_j \rightarrow_{\text{max}} S_4 \ b \ (c_1)_{\tau_j} \mid 1 \leq j \leq N \}.$$  

We argue that our approach has positive consequences, both at the conceptual and practical (implementation) level.

4 Minimum Spanning Tree (Sync)

We further illustrate our extensions with a P system solution for another famous (but more complex) problem, the distributed construction of a minimum spanning tree (MST). The solution minimizes the total communication cost between nodes (i.e. cells). Note that, in our case, this problem requires an additional mechanism, to reify arc weights.

This optimization problem has many applications in a wide variety of domains and has been solved in a multitude of models, including ant colonies and DNA computing, but, as far as we know, not yet in P systems. Its distributed version may offer additional insights into how biological colonies organize their activities efficiently.

5 Asynchronous P Systems

The traditional P system model is synchronous, i.e. all cells’ evolution is controlled by a single global clock. P systems with various asynchronous features have been investigated by recent research [18, 3, 7, 4, 6, 5, 17, 22, 27, 31]. We are looking for similar but simpler definitions, closer to the standard definitions used in distributed algorithms [23, 30]. We are interested to model fundamental and challenging algorithms and to assess the merits of such modeling exercise. We hope that the feedback of this experience will be useful to the larger P system community.

Here, we further elaborate the ideas first proposed in our previous paper [1]. In contrast to the synchronous case, fully asynchronous P systems are characterized by the absence of any system clock, let alone a global one; however, an outside observer may very well use a clock to time the evolutions. Our approach, based on classical notions in distributed algorithms, as presented by Tel [30], does not require any change in the static descriptions of P systems and only their evolutions differ (i.e. just the underlying P engine works differently):

1. for each cell, each step starts after a random step delay $t$ (after the preceding step);
2. for each cell, each step, once started, takes zero time (i.e. it occurs instantaneously; although a small execution delay might at times be more realistic);
3. for each message, its delivery delay $t$ is random, either from its origin, or, more realistically, after the previous message over the same channel (i.e. arc).

We typically assume that messages sent over the same arc arrive in FIFO order (queue)—but one could also consider arrival in arbitrary order (multiset instead of queue). Synchronous P systems can be considered as a special case of asynchronous P systems, where all step and delivery delays are one, i.e. $t = 1$.

For the purpose of time complexity, the time unit is chosen to be greater than any step or delivery delay, i.e. all such delays are real numbers in the closed unit interval, i.e. $t \in [0, 1]$. The runtime complexity of an asynchronous system is the supremum over all possible executions.

We briefly discuss several fundamental notions and properties related to asynchronous systems [23, 30], such as causality, liveness, safety, fairness, which suggests further directions of study, of proof techniques for asynchronous P systems.

6 Leader Election and Minimum Spanning Tree (Async)

As a sequel to the examples of our previous paper [1], we validate our proposals for asynchronous P systems with a solution for leader election and a solution for minimum spanning trees, which is one of the classic and challenging problems in asynchronous distributed algorithms, first solved by Gallager et al. [19].

7 Parallel Stereo Matching

Image processing offers many opportunities for parallel modelling, but, with a few notable exceptions, mostly from the Seville group, such as Carnero et al. [2], has not yet attracted much attention from the P systems community.

To finalize this presentation, we briefly present an image processing application, detailed in our forthcoming paper [21]. We designed a massively parallel synchronous P model for implementing a critical part of the dynamic programming stereo matching algorithm proposed by Gimel’farb [20]. Our model processes in parallel all potentially optimal similarity scores that trace candidate decisions, for all the disparities associated with each current $x$-coordinate. The theoretical performance of our P model is conceptually comparable to that of a physical parallel processor with an unlimited number of processing elements.

The modelling exercise has enabled us to identify a small bug in an existing practical implementation (in the C programming language), but also, more importantly, to refactor it, following our cell structure. The result is a more robust and flexible version, which allows us to fine tune its parameters and enhance its capabilities, without rewriting it from scratch. We think that our modelling exercise would have been practically impossible without some of the additional ingredients discussed in this paper.

Figure 2 shows, in order, a monocular left image, a monocular right image and their true disparity map (the ground truth). Figure 3 shows computed disparity maps: left—by the old program; right—by our refactored and better tuned program.
8 Conclusions

We suggest a few simple criteria for assessing the merits of such modelling approaches, e.g., shorter and crisper descriptions, but without sacrificing the runtime performance, when compared to other approaches, such as state-of-art pseudo-code. We believe that our proposed additional ingredients have proved their usefulness in refactoring existing realistic systems and could be interesting to the larger P system community.

Acknowledgments

The author wishes to acknowledge the contributions of M.J. Dinneen, G.L. Gimel’farb, Y.-B. Kim, J. Morris, S. Ragavan, H. Wu and the assistance received via the University of Auckland FRDF grant 9843/3626216.

References


Variants of Distributed P Automata and the Efficient Parallelizability of Languages*  

Abstract  

György Vaszil  

Computer and Automation Research Institute  
Hungarian Academy of Sciences  
Kende u. 13-17, 1111 Budapest, Hungary  
vaszil@sztaki.hu  

1 Introduction  

Although P systems are distributed parallel computing devices themselves, their high computational capacity is mostly due to their ability to work with an exponential amount of workspace in polynomial time, see [6] for an overview.  

A different approach to computing with P systems in a distributed way was proposed in [5] where the notion of a distributed P automaton, or dP automaton in short, was introduced. Such a dP automaton consists of components which process different parts of the input that is split into several pieces, the different pieces being simultaneously operated on by the different components of the system which might also communicate with the others in order to provide the result of the computation.  

A language is parallelizable if its words can be recognized by a distributed P automaton in such a way, that the parts processed by the different components are of roughly the same size. Such a parallelization of a language is efficient, if the computation of the distributed P automaton uses less computational resources than the any non-distributed variant.  

In [5], the formal notion of the efficiency of parallelization was introduced in a way which essentially says that efficiency is implied by the existence of a distributed computation which is k times faster than any non-distributed one (for some k > 1) while the amount of communication between the components is constant.  

As P automata read multiset sequences from the environment, to recognize string languages, that is, to be able to speak of accepted or rejected words (which are strings of symbols of some alphabet), P automata need to employ mappings which map the sequences of input multisets to symbol sequences (strings) over the alphabet of the language in question. This mapping, which we call input mapping in the sequel, can be chosen in several different ways.  

In the following we first examine how the choice of the possible input mappings influences the efficient parallelizability of P automata languages (see also  

* Supported in part by the Hungarian Scientific Research Fund, “OTKA”, grant no. K75952.
[7]), and demonstrate the existence of a linear speedup property for certain P automata variants. Then we propose modifications in the above described notion of efficient parallelizability which would make it less restrictive (by relaxing the requirement of a constant amount of communication between the components), but also more precise (by allowing the classification of the different speed improvement rates other than just the constant one).

2 Preliminaries and definitions: P automata and distributed P automata

Let $V$ be an alphabet, let $V^*$ be the set of all words over $V$ where $\varepsilon$ denotes the empty word, let $|w|$ denote the length, the number of symbols of $V$ in $w \in V^*$, and let $|w|_a$ denote the number of occurrences of the letter $a \in V$ in $w$. The set of non-negative integers is denoted by $\mathbb{N}$.

A multiset is a mapping $M : V \to \mathbb{N}$ where $V$ is a set of objects and $M$ assigns to each object its multiplicity. The empty multiset is denoted by $\emptyset$. A multiset over $V$ can be represented as any string $w \in V^*$ where $|w|_a = M(a)$ for all $a \in V$. Note that all permutations of a given string represent the same multiset. In the following we use this string representation for multisets, but to avoid confusion, and to distinguish between strings of objects and sequences (strings) of multisets, we designate the set of finite multisets over a set $V$ by $V^{(\ast)}$, and the set of their sequences by $(V^{(\ast)})^\ast$.

The classes of recursively enumerable, context-sensitive, context-free, linear, and regular languages are denoted by RE, CS, CF, LIN, and REG, respectively.

A P automaton, introduced in [3], is a system $\Pi = (V, \mu, P_1, \ldots, P_n, c_0, \mathcal{F})$ where $V$ is an object alphabet, $\mu$ is a membrane structure, $P_i$, $1 \leq i \leq n$ are sets of antiport rules, $c_0 = (w_1, \ldots, w_n)$ is the initial configuration with $w_i \in V^*$, $1 \leq i \leq n$ being the initial contents of the $i$th region, and $\mathcal{F}$ is a set of accepting configurations of the form $(v_1, \ldots, v_n)$, $v_i \in V^*$, $1 \leq i \leq n$ where $\mathcal{F}$ is given as $E_1 \times \ldots \times E_n$, $E_i \subseteq V^*$, such that $E_i$ is either finite, or $E_i = V^*$, $1 \leq i \leq n$.

The configurations of the P automaton are changed by applying the rules in the maximal parallel manner, that is, as many rules are applied in each region as possible.

For two configurations $c, c' \in (V^*)^n$, we say that $c' \in \delta(u, c)$ if $\Pi$ enters configuration $c'$ from configuration $c$ by applying its rules while reading the input $u \in V^\circ$, that is, if $u$ is the multiset that enters the system through the skin membrane from the environment while the configuration $c$ changes to $c'$. The sequence of configurations obtained this way is called a computation. If it ends in a final configuration from $\mathcal{F}$, then the sequence of multisets entering the system from the environment in each step of the computation is called an accepted multiset sequence. Thus, $v_1, \ldots, v_s$, $v_i \in V^*$, $1 \leq i \leq s$, is a multiset sequence accepted by $\Pi$ if there are $c_0, c_1, \ldots, c_s \in (V^*)^n$, such that $c_i \in \delta(v_i, c_{i-1})$, $1 \leq i \leq s$, and $c_s \in \mathcal{F}$.
Let $\Pi$ be a P automaton as above, and let $f$ be a mapping $f : V^* \rightarrow 2T^*$ for some finite alphabet $T$. We call $f$ the input mapping of $\Pi$. Let us assume that $f$ is nonerasing, that is, $f(u) = \{\varepsilon\}$ for some $u \in V^*$, if and only if $u = \emptyset$.

The language over $T$ accepted by $\Pi$ with respect to $f$ is defined as

$$L(\Pi, f) = \{f(v_1) \ldots f(v_s) \mid v_1, \ldots, v_s \text{ is an accepted multiset sequence of } \Pi\}.$$  

Note the influence of the choice of $f$ in the definition above on the language accepted by the P automaton. It should be chosen in such a way that the power of the model lies in the underlying P system and not in the mapping $f$.

Remark 1. If input mappings with erasing, that is, with $f(u) = \{\varepsilon\}$ for some $u \in V^*$, $u \neq \emptyset$ are used, then we obtain so called extended P automata which are able to characterize RE with systems of a relatively simple structure (see Chapter 6.4 in [2] for details).

A distributed P automaton, or dP automaton, introduced in [5] is a construct $d\Pi = (V, \Pi_1, \ldots, \Pi_k, R)$ where $V$ is a finite set of objects, $\Pi_i$, $1 \leq i \leq k$ are the components of the system with

$$\Pi_i = (V, \mu_i, P_{i,1}, \ldots, P_{i,m_i}, c_{i,0}, F_i), \quad 1 \leq m_i,$$

being P automata as defined above having the skin membranes labeled by $(i,1)$, and $R$ is a finite set of inter-component communication rules of the form $((i,1), u/v, (j,1))$ with $u, v \in V^*$, $1 \leq i, j \leq k$, $1 \neq j$. The initial configuration of the dP automaton is $c_0 = (c_{1,0}, \ldots, c_{k,0})$.

The language $L \subseteq T^*$ accepted by a dP automaton consists of words of the form $w_1w_2\ldots w_k$ where $w_i \in T^*$ are strings accepted by the component $\Pi_i$, $1 \leq i \leq k$, during a successful computation, that is, one that starts in $c_0$ and ends in one of the final configurations of $F = (F_1, \ldots, F_k)$.

More formally, let $f = (f_1, \ldots, f_k)$ be a mapping $f : (V^*)^k \rightarrow (2T^*)^k$ with $f_i : V^* \rightarrow 2T^*$, $1 \leq i \leq k$, being nonerasing (see Remark 1 above), and let

$$L(d\Pi, f) = \{w_1 \ldots w_k \in T^* \mid w_i \in f_i(v_{i,1}) \ldots f_i(v_{i,s_i}), \quad 1 \leq i \leq k,$$

where $v_{i,1}, \ldots, v_{i,s_i}$ is an accepted multiset sequence of the component $\Pi_i\}.$

### 3 The power of P automata and the influence of the input mapping

In order to study the effect that the different types of input mappings have on the properties of (distributed) P automata, we recall the two basic types of mappings studied in [7]. Let $f : V^* \rightarrow 2T^*$, for some alphabets $V$ and $T$, and let the mapping $f_{\text{perm}}$ and the class of mappings TRANS be defined as follows:

- $f = f_{\text{perm}}$ if and only if $V = T$ and for all $v \in V^*$, we have $f(v) = \{a_1a_2\ldots a_s \mid |v| = s$, and $a_1a_2\ldots a_s$ is a permutation of the elements of $v\}$;
– $f \in \text{TRANS}$ if and only if for any $v \in V^{(*)}$, we have $f(v) = \{w\}$ for some \( w \in T^* \) which is obtained by applying a finite transducer to the string representation of the multiset $v$, (as $w$ is unique, the transducer must be constructed in such a way that all string representations of the multiset $v$ as input result in the same $w \in T^*$ as output, and moreover, as $f$ should be nonerasing, the transducer produces a result with $w \neq \varepsilon$ for any nonempty input).

Let the class of languages accepted by P automata or distributed P automata having $k$ components with $f_{\text{perm}}$ or with an input mapping from the class TRANS be denoted by $L_X(Y)$ where $X \in \{\text{PERM, TRANS}\}$, $Y \in \{\text{PA, dPA}_k\}$, respectively. If we do not wish to emphasize the number of components of dP automata, we might omit the subscript $k$ from $\text{dPA}_k$.

As already mentioned above, the type of mappings used to map the sequence of input multisets to a sequence of terminal letters has great influence on the properties of languages accepted by (distributed) P automata. It is shown, for example, in [4] that the linear language of palindromes cannot be accepted by P automata with $f_{\text{perm}}$, that is,

$$L = \{wcw^{-1} \mid w \in \{a, b\}^*, w^{-1} \text{ is the reverse of } w\} \not\in L_{\text{PERM}}(\text{PA}).$$

As $L_{\text{PERM}}(\text{PA})$ contains non-context-free context-sensitive languages ($\{a^n b^n c^n \mid n \geq 1\}$ for example), this means that it is incomparable with LIN and CF. (Although it contains all regular languages, [4].)

On the other hand, as we know from [1] (see also [2]), any context-sensitive language can be accepted with P automata using a very simple input mapping from the class TRANS, thus,

$$L_{\text{TRANS}}(\text{PA}) = \text{CS}.$$

**Remark 2.** Consider a P automaton $\Pi$ with objects from $V$, and any nonerasing mapping $f : V^* \to 2^{T^*}$ for some alphabet $T$. The results from [1, 2] also imply that $f$ can be of any kind, as long as it is not more complex than linear space computable (in the Turing machine model), the accepted language is context sensitive, that is, $L(\Pi, f) \in \text{CS}$. 

### 4 The parallelizability of languages

The notion of the parallelizability of languages was introduced and first studied for languages of $L_{\text{PERM}}(\text{PA})$ in [5]. In [7] also the class $L_{\text{TRANS}}(\text{PA})$ was considered. In the following, we first point out the difficulties of using the particular notion of parallelizability introduced in [5] for systems having input mappings from TRANS, then propose a way to overcome these by using a more general notion of parallelizability than the previous one.

A computation accepting a string $w \in T^*$ by a dP automaton as above with components $\Pi_i$, $1 \leq i \leq k$, is balanced, if $w = w_1 \ldots w_k$ for $w_i \in L(\Pi_i, f_i)$, such
that $-1 \leq |w_j| - |w_l| \leq 1$ for any $1 \leq j, l \leq k$, where $L(\Pi_i, f_i)$ denotes the set of words read by the component $\Pi_i$ during a successful computation of the dP automaton $d\Pi$.

According to [7] (based on [5]), a language $L$ is $(k, l, m)$-efficiently parallelizable with respect to a class of mappings $F$, for some $k, m > 1$, $l \geq 1$, if $L$ can be accepted with balanced computations of a dP automaton $d\Pi$ with $k$ components, such that $L = L(d\Pi, f)$ for some $f \in F$ with $\text{Com}(d\Pi) \leq l$, and moreover, for all P automata $\Pi$ and $f' \in F$ such that $L = L(\Pi, f')$,

$$\lim_{x \in L, |x| \to \infty} \frac{\text{time}_{\Pi}(x)}{\text{time}_{d\Pi}(x)} \geq m$$

where $\text{time}_X(x)$ denotes the number of computational steps that a device $X$ needs to accept the string $x$, and where Com is a communication measure denoting the maximal amount of communication (measured in some reasonable way, for example, as the number of communication steps, the number of applied communication rules, or the number of communicated symbols) between the components of a dP automaton during an accepting computation.

Thus, the above defined notion of efficient parallelizability of $L$ expresses the fact that $L$ can be accepted by a dP automaton which uses a finite amount of communication ($\text{Com}(d\Pi) \leq l$) while working about $m$ times faster than any (non-distributed) P automaton which accepts $L$. In other words, the parallelization is efficient, if the dP automaton achieves a linear speedup in computational time compared to any instance of the sequential variant.

There are P automata, however, which (similarly to the linear speedup property of Turing machines) can be modified to operate faster with any constant factor while accepting the same language, and unfortunately, this speedup property might also depend on the type of input mapping used.

5 \hspace{1cm} The linear speedup of P automata and the efficiency of parallelization

Let us consider the case of regular languages first, see [5]. Obviously, there are regular languages with the property that the order of no two adjacent symbols can be exchanged, thus, a P automaton which accepts such a language and uses the input mapping $f_{\text{perm}}$ has to read all input letters in a different computational step. This means that the number of steps of a (non-distributed) P automaton cannot be less than the length of the input. This is not the case, however, with dP automata where each component reads just a portion of the input, thus, the number of computational steps might be less than the input length.

\textbf{Theorem 1} ([5]). \textit{There are $(k, l, m)$-efficiently parallelizable regular languages with respect to the mapping $f_{\text{perm}}$ for some $k, m > 1$, $l \geq 1$.}

On the other hand, considering P automata with input mappings from TRANS, any system accepting a regular language can be simulated by an other one which is faster by a constant factor.
Theorem 2 ([7]). For any language \( L \in \text{REG} \) and constant \( c > 0 \), there exists a P automaton \( \Pi \) such that \( L = L(\Pi, f) \) for some \( f \in \text{TRANS} \), and for any \( w \in L \) with \( |w| = n \) it holds that \( \text{time}(w) \leq c \cdot n \).

This means that no distributed P automaton can be faster than all the non-distributed variants, thus we have the following.

Corollary 3 ([7]) There are no \((k, l, m)\)-efficiently parallelizable regular languages with respect to the class of mappings \( \text{TRANS} \) for any \( k, m > 1, l \geq 1 \).

In order to present a more general statement, we need the following definition. A language \( L \) is real-time recognizable by a P automaton \( \Pi \) if \( L = L(\Pi, f) \) for some input mapping \( f \), and \( \Pi \) reads a nonempty input multiset in each step of any computation accepting the words of \( L \).

Theorem 4. Let \( L \) be a language which is real-time recognizable by a P automaton \( \Pi_1 \), such that \( L = L(\Pi_1, f) \) for some \( f \in \text{TRANS} \), and let \( c > 0 \) be a constant. Then there exists a P automaton \( \Pi_2 \), such that \( L = L(\Pi_2, f') \) for some \( f' \in \text{TRANS} \), and for any \( w \in L \) it holds that \( \text{time}(w) \leq c \cdot \text{time}(w) \).

To see this, note that for any P automaton \( \Pi \) of a membrane structure with \( n \) membranes and with an object alphabet \( V \), we construct a \( \Pi' \) with one membrane and object alphabet \( V' = V \cup \{a_1 | a \in V, 1 \leq i \leq n\} \). The work of \( \Pi \) can be simulated by \( \Pi' \) in the following way: The fact that \( a \in V \) is present in region \( i \geq 2 \) of \( \Pi \) is reflected by \( a_i \in V' \) being present in the single region of \( \Pi' \) and the presence of either \( a \) or \( a_1 \) for \( i = 1 \) (if in \( \Pi \), \( a \) is imported from the environment, then it is represented by \( a \) in \( \Pi' \), if it is exported from an inner region, then it is represented by \( a_1 \)). For example, the effect of the antiport rule \((a, \text{in}; b, \text{out})\) of \( \Pi \) in a region \( i \geq 2 \) can be simulated by the rules \((a_i, b_j, \text{in}; a_j, b_i, \text{out})\) in the skin region of \( \Pi \), where region \( j \) is the parent region of \( i \), or an antiport rule \((a, \text{in}; b, \text{out})\) of \( \Pi \) in region 1 can be simulated by the rules \((a, \text{in}; b_1, \text{out})\), \((a, \text{in}; b, \text{out})\). The P automaton \( \Pi_2 \) accepts \( L \) in real-time, which means that any input multiset \( u' \in V'' \) of \( \Pi' \) contains a nonempty sub-multiset \( u \in V'' \). Therefore, by defining \( f' \) as \( f'(u') = f(u) \) where \( u \in V'' \) is the above mentioned sub-multiset of \( u' \in V''' \), we have that \( L(\Pi', f') = L(\Pi, f) \) and \( \Pi' \) is a system with one membrane.

Now we combine the antiport rules of the P automaton having one membrane in such a way that the new rules simulate the execution of two consecutive steps of the old ones (with further modification in the alphabet to be able to separate the symbols which enter in the first and the second simulated step). With proper modifications also in the input mapping, we obtain a P automaton \( \Pi'' \), such that \( L(\Pi'', f'') = L(\Pi', f') \), and moreover, for any \( w \in L \), \( \text{time}(w) \leq \frac{\text{time}(w)}{2} \).
6 Efficient parallelizability with greater speedup and more than a constant amount of inter-component communication

As we have seen above, the computations of P automata can be made faster by any constant factor in many cases, thus sometimes a linear speedup of the distributed computation cannot be enough to achieve efficient parallelization. On the other hand, there might also be dP automata which are “polynomially” or “exponentially” more efficient than their non-distributed counterparts. Moreover, a dP automaton may accept a language faster than the non-distributed variant even if the amount of communication between the components does not remain under a constant bound. To incorporate these features, we propose the following definition of (efficient) parallelizability.

A language \( L \) is \((k, t(n))\)-parallelizable with respect to the class of mappings \( F \), for some \( k > 1 \) and a function \( t : \mathbb{N} \rightarrow \mathbb{N} \), if there is a dP automaton \( d\Pi \) with \( k \) components and \( L(d\Pi, f) = L \) for some \( f \in F \), and there is an \( n_0 \in \mathbb{N} \), such that for any \( w \in L \), \( |w| = n \), \( n > n_0 \), we have

\[
\frac{\text{time}_H(w)}{\text{time}_{d\Pi}(w) + \text{Com}_{d\Pi}(w)} \geq t(n)
\]

for any non-distributed P automaton \( H \) with \( L(H, f') = L \), \( f' \in F \), where \( \text{Com}_{d\Pi}(w) \) denotes the amount of communication realized by \( d\Pi \) during the computation on \( w \).

The language \( L \) is efficiently parallelizable if it is \((k, t(n))\)-parallelizable for some \( t \) with \( t(n) > 1 \) if \( n \geq n_0 \) for some \( n_0 \in \mathbb{N} \).

Consider the following example.

**Example 1** Let \( L = \{ww \mid w \in \{a, b, c\}^*, \; |w| = 3t \; \text{for some} \; t \in \mathbb{N}\} \). As there are words in \( L \) in which no two adjacent symbols can be exchanged in such a way that the result is still in \( L \), any non-distributed P automaton with input mapping \( f_{\text{perm}} \) needs at least \( n = 6t \) steps to recognize the words of length \( n \).

On the other hand, we can construct a distributed P automaton with input mapping \( f_{\text{perm}} \) and two components which work by reading the first and second half of the input simultaneously, and sending messages to each other after each triple of symbols which they read in order to make sure that the read triples are the same.

This way of functioning requires \( \frac{1}{2}n = 3t \) computational steps and \( t \) communication steps in addition (each transmitting a constant amount of information), giving an overall complexity of \( 4t = \frac{3}{2}n \) which makes the above described parallelization of \( L \) efficient.

Thus, as we have seen in the example above, there are languages which are parallelizable in such a way that the distributed recognition of these words uses less computational resources than any non-distributed computation, so they are efficiently parallelizable, but not considered in [5, 7] because of the violation of the constant communication requirement.
7 Conclusion

In the talk we explore these topics in more detail with more examples and more elaborated arguments.

References

Regular Papers
Quantitative Causality in Membrane Systems

Oana Agrigoroaiei and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science
Blvd. Carol I no.8, 700505 Iași, Romania
oanaag@iit.tuiasi.ro gabriel@info.uaic.ro

Abstract. We define and study local and global causality in terms of multisets of objects and of multisets of rules. We relate the two notions of causality, and provide characterizations for each of them. We give an inductive method of obtaining the global causes of a multiset of objects, and use the global causes of a multiset \( v \) to find all the multisets \( u \) from which \( v \) can be obtained in a single evolution step.

1 Introduction

The notion of causality for complex systems involving parallelism and non-determinism is described in the existing literature in various modes, ranging from that given by temporal ordering to placing certain “markers” on events [4]. For membrane systems, the main existing approaches are provided by translating them into other formalisms such as Petri nets [7], event structures [5] or event automata [10]. Besides lacking a direct definition in terms of membrane systems [8, 9], such approaches neglect quantitative aspects involved in membranes evolution. An event or a Petri net transition is generally considered to be the application of a single rule, and cannot provide causal information with respect to a requested quantity of resources. We have initiated a quantitative approach to causality in [1] by defining both rule-based and object-based event structures. The causality inherent in object-based structure was defined for specific multisets of objects. In this paper we define causality for any multiset of objects directly, without reducing membranes to a second formalism. We start with a local definition which is related to a certain evolution step. We also give a global definition for any multiset, which we prove to be in direct correspondence with the local one. We work at both local and global level since some results can only be obtained at a local level (Proposition 13) and have no correspondent at a global level, while others (Theorem 11) take place only at a global level.

We use the global notion of causality to characterize multisets \( u \) which, after a step \( u \xrightarrow{F} u' \), evolve to a multiset containing a given multiset \( v \) \( (v \leq u') \). This characterization is static and does not require actually the execution of the evolution steps; it only requires computing the global causes of \( v \). As far as we know, no other above mentioned approach to causality can solve such a problem (see Example 17).
2 Multisets in Membrane Systems

We work with transition P systems with only one membrane, which are equivalent to transition P systems with multiple membranes and without dissolution [3, 6]. Generally, a transition P system consists of a hierarchy of nested membranes, placed inside a distinguishable membrane called skin. Each membrane can contain multisets of objects, evolution rules and other membranes. We restrict ourselves to the case of systems with only one membrane, which are defined as follows.

A membrane system with only one membrane is a tuple $\Pi = (O, R)$, where

- $O$ is an alphabet of objects;
- $R$ is a finite set of rules, each rule having the form $x \rightarrow y$, where $x, y$ are multisets of objects and $x$ is non-empty.

We use multisets of objects to represent resources available or being produced in the membrane. A multiset of objects (i.e., over $O$) found initially in the membrane evolves by having rules from $R$ applied to it; by rule application we understand that the left hand side $x = \text{lhs}(r)$ of a rule $r : x \rightarrow y$ is being subtracted from $u_0$, if possible, and after this the right hand side $y = \text{rhs}(r)$ of the rule is added. In this way, rule application models chemical reactions. An evolution step consists of several rule applications done concurrently, in such a manner that objects obtained from rule application are not consumed by other rules, but are instead left in the membrane.

One of the central notions of the membrane systems framework is that of finite multiset, which we recall here, before giving a formal description of membrane evolution. All multisets we use in this paper are considered finite. A finite multiset over a set $S$ can be defined either as an equivalence class of a finite word over $S$ or as a function from $S$ to the set of natural numbers $\mathbb{N}$. In this paper we use the function definition of a multiset. When describing a multiset characterized by, for example, $w(s) = 4$, $w(t) = 2$ and $w(s') = 0$ for $s' \in S \setminus \{s, t\}$, we use the representation $4s + 2t$. To each multiset $w$ we associate its support, denoted by $\text{supp}(w)$, which contains those elements of $S$ which have a non-zero image. A multiset is called non-empty if it has non-empty support. We denote the empty multiset by $0$. We sometimes overload the set notation to multisets by using $s \in w$ instead of $w(s) \geq 1$.

The sum of two multisets $w, w'$ over $S$ is the multiset $w + w' : S \rightarrow \mathbb{N}$, defined by $(w + w')(s) = w(s) + w'(s)$. For two multisets $w, w'$ over $S$ we say that $w$ is contained in $w'$ if $w(s) \leq w'(s)$ for all $s \in S$, and we denote this by $w \leq w'$. If $w \leq w'$, we can define $w' - w$ by $(w' - w)(s) = w'(s) - w(s)$.

We also employ other operations over multisets, derived from set operations.

Definition 1. For multisets $x, y$ over a set $S$ we define the multisets $x \cap y, x \cup y$ and $x \setminus y$ over $S$ by:

- $(x \cap y)(a) = \min\{x(a), y(a)\}$;
- $(x \cup y)(a) = \max\{x(a), y(a)\}$;
\((x\setminus y)(a) = \max\{x(a) - y(a), 0\}\).

We present some useful properties of operations over multisets, which we use throughout the paper.

**Proposition 2.** Let \(x, y, z\) be multisets over a set \(S\). Then:

1. \(x\setminus y = x - x \cap y\) and \(x \cup y = x + y - x \cap y\);
2. \((x \cup y) \cap z = (x \cap z) \cup (y \cap z)\) and \((x \cap y) \cup z = (x \cup z) \cap (x \cup z)\);
3. if \(x \leq y\) then \(x \cap z \leq y \cap z, x \cup z \leq y \cup z\) and \(z \setminus x \geq z \setminus y\);
4. \((x \cup y) \setminus z = (x \setminus z) \cup (y \setminus z)\);
5. \(x \leq y + z\) iff \(x \leq x \cap y + x \cap z\);
6. \(x \leq y + z\) iff \(x \setminus z \leq y\);
7. \((x \setminus y) \setminus z = x \setminus (y + z)\);
8. \((x + y) \setminus z = x \setminus (y + z)\).

The following definition gives the operational semantics of a system with one membrane.

**Definition 3.** Let \(F\) be a multiset of rules (i.e., over \(R\)). We denote by \(\text{lhs}(F)\) the multiset of objects obtained by adding the left hand sides of all the rules in \(F\), considered with their multiplicities: \(\text{lhs}(F) = \sum_{r \in R} F(r) \cdot \text{lhs}(r)\). The multiset \(\text{rhs}(F)\) is defined similarly.

Let \(u, u', w\) be multisets of objects. Then:

- \(w\) is called stable if there is no rule \(r \in R\) which can be applied to \(w\): \(\not\exists r \in R\) such that \(\text{lhs}(r) \leq u\);
- \(F\) is called maximally parallel with respect to \(u\) if \(\text{lhs}(F) \leq u\) and \(u - \text{lhs}(F)\) is stable;
- \(u\) is said to evolve to \(u'\) through \(F\) (denoted by \(u \overset{F}{\rightarrow} u'\)) whenever \(F\) is maximally parallel with respect to \(u\) and \(u' = u - \text{lhs}(F) + \text{rhs}(F)\).

### 3 Local and Global Causality

We start by defining *covers* for multisets of objects which have been produced during the evolution of the system. Then we refine the notion of cover to describe the cause of such a multiset.

A cover \((G, w)\) for a multiset \(v\) is defined with respect to a certain evolution step \(u \overset{F}{\rightarrow} u'\) in which \(v\) is obtained, namely for \(v\) being a part of the multiset \(u'\) \((v \leq u')\). It is formed from a multiset of rules \(G\) and a multiset of objects \(w\) such that they both contribute to the occurrence of \(v\). The main idea is that \(G \leq F\) is a part of the rules used in the evolution step and \(w \leq u - \text{lhs}(F)\) is a part of the objects which are not consumed by rules, such that \(v \leq w + \text{rhs}(G)\). In other words, \(v = w + (v - w)\) is obtained from \(u\) in one evolution step \(u \overset{F}{\rightarrow} u' = u - \text{lhs}(F) + \text{rhs}(F)\), such that \(w\) are objects which are not consumed by any rule and \(v - w\) are objects produced by rules in \(G\).

The following definition formalizes the notion of (local) cover.
Definition 4. Consider multisets $F, G$ of rules and multisets $u, v, w$ of objects. A pair $(G, w)$ is called a cover of $v$ with respect to $u$ and $F$ whenever the following hold:

- $F$ is maximally parallel with respect to $u$;
- $w \leq v$ and $w \leq u - \text{lhs}(F)$;
- $v - w \leq \text{rhs}(G)$.

We let $\text{Cover}_{u,F}(v)$ denote the set of covers of $v$ with respect to $u$ and $F$. We also use the term local cover when we refer to such a cover.

The set $\text{Cover}_{u,F}(v)$ of covers is non-empty if and only if $u \xrightarrow{F} u - \text{lhs}(F) + \text{rhs}(F) \geq v$. When this condition holds, we say that $v$ is obtainable from $u$ through $F$.

Example 5. Consider the membrane system with alphabet $O = \{x, y, a, b\}$ and rules $r_1 : x \rightarrow a + b, r_2 : y \rightarrow b$. Let $u = 2x + y + a + b; F$ can only be $2r_1 + r_2$.

Examples of covers of $2a$ are $(2r_1, a), (r_1 + r_2, a)$. Intuitively, such covers contain more information than needed to produce those $2a$ (they each produce $3a$).

Another cover for $2a$ is $(2r_1, 0)$, which is smaller than $(2r_1, a)$. We see that we cannot find a cover smaller than $(2r_1, 0)$; trying to remove a $r_1$ from it leads to no longer having cover properties. The idea we promote is that the cause of a multiset of objects $v$ ($2a$ in our example) is given by a multiset of rules $G$ and a multiset of objects $w$ which are minimal (for a certain minimality condition) and still make $v$ appear.

Proposition 6. Consider $u, v$ such that $v$ is obtainable from $u$ through $F$. If $(G, w)$ is a cover of $v$ with respect to $u$ and $F$, then so is $(G, v \setminus \text{rhs}(G))$.

Consequently, if a pair $(G, w)$ is a minimal element of $\text{Cover}_{u,F}(v)$ with respect to the product order then $w = v \setminus \text{rhs}(G)$.

Proof. Let $(G, w)$ be an element of $\text{Cover}_{u,F}(v)$. Since $v \leq w + \text{rhs}(G)$, it follows (by using Proposition 2) that $v \setminus \text{rhs}(G) \leq w$. We prove that $(G, v \setminus \text{rhs}(G))$ is also a cover for $v$ with respect to $u$ and $F$. First, $v \setminus \text{rhs}(G) \leq w \leq u - \text{rhs}(F)$; second, $v - v \setminus \text{rhs}(G) = \text{rhs}(G) \cap v \leq \text{rhs}(G)$.

If $(G, w)$ is a minimal cover, it follows that $w = v \setminus \text{rhs}(G)$ since $(G, v \setminus \text{rhs}(G))$ is a cover which is smaller or equal to $(G, w)$.

Proposition 6 states that a minimal cover $(G, w)$ of a multiset depends only on the multiset $G$ of rules. Let $\text{Cause}_{u,F}(v)$ denote the set of rules multisets $G$ for which $(G, v \setminus \text{rhs}(G))$ is a minimal element of $\text{Cause}_{u,F}(v)$ with respect to the product order on this set.

Example 7. In Example 5, for $v = 2a$ and $u = 2x + y + a + b$, we have minimal covers $(2r_1, 0)$ and $(r_1, a)$, so $\text{Cause}_{u,F}(v) = \{r_1, 2r_1\}$. Indeed, in the evolution step $u \xrightarrow{2r_1 + r_2} 3a + 4b$, the multiset $v = 2a$ is produced either by a rule $r_1$ and the multiset $a = v \setminus \text{rhs}(r_1)$ which is not consumed by any rule or it is produced by two rules $r_1$ and by no multiset which is not consumed...
(0 = v \text{rhs}(2r_1)). If we consider u′ = 2x + y, the only possible evolution step is 2x + y \xrightarrow{2r_1+r_2} 2a + 3b and there is only one minimal cover for v, namely (2r_1, 0) which yields \text{Cause}_{u′,F}(v) = \{2r_1\}. We see thus how the possible causes vary when the multiset u is changed. To see how the causes change in the event that the multiset of rules F changes, consider the membrane system with alphabet \( O = \{x, y, a, b, c\} \) and rules \( r_1 : x \rightarrow a + 2b, r_2 : x + y \rightarrow b + c \). Let \( u = x + y, v = b \) and \( F = r_1, F′ = r_2 \). Then \( \text{Cause}_{u,F}(v) = \{r_1\} \) and \( \text{Cause}_{u,F′}(v) = \{r_2\} \).

In this example we see that the same resource \( v \) obtained from two different evolution steps starting from the same \( u \) can have different causes with respect to each evolution step.

We can also define covers and causes at a global level, without restricting ourselves to a particular \( u \) and \( F \) through which \( v \) is obtained. We show that the global definitions are, as expected, generalizations of the local ones.

**Definition 8.** Let \( \text{Cover}(v) \) be the set of pairs \((G, w)\) of rules multisets \( G \) and stable multiset \( w \) of objects such that

\[
    w \leq v \leq w + \text{rhs}(G).
\]

Let \( \text{Cause}(v) \) be the set of rules multisets \( G \) for which there exists \( w \) such that \((G, w)\) is a minimal element of \( \text{Cover}(v) \) with respect to the product order.

We refer to the covers and causes defined now as global covers and causes. We prove that each global cover or cause of \( v \) is also a local cover or cause of \( v \) with respect to certain \( u \) and \( F \). Reversely, each local cover or cause of \( v \) with respect to some \( u \) and \( F \) is also a global cover or cause. This statement is summarized in the following proposition.

**Proposition 9.** Let \( v \) be a multiset of objects and let \( F \) denote the family of pairs \((u, F)\) for which \( v \) is obtainable from \( u \) through \( F \). Then

\[
    \text{Cover}(v) = \bigcup_{(u,F) \in F} \text{Cover}_{u,F}(v) \quad \text{and} \quad \text{Cause}(v) = \bigcup_{(u,F) \in F} \text{Cause}_{u,F}(v).
\]

**Proof.** We start by proving the first equality. Consider \((G, w)\) in \( \text{Cover}(v) \), and let \( u = w + \text{lhs}(G), F = G \). Then \((G, w) \in \text{Cover}_{u,F}(v)\). Reversely, if \((G, w) \in \text{Cover}_{u,F}(v)\) for some \((u, F) \in \mathcal{F}\), then \( w \) is stable and \( w \leq v \leq w + \text{rhs}(G) \).

For the second equality, let \( G \in \text{Cause}(v) \). Then there exists a stable \( w \) such that \((G, w)\) is a minimal cover of \( v \). As we just proved, \((G, w) \in \text{Cover}_{u,F}(v)\) for \( u = w + \text{lhs}(G) \) and \( F = G \). We prove that \( G \in \text{Cause}_{u,F}(v) \). We assume there exists \((H, z) \in \text{Cover}_{u,F}(v)\) such that \((H, z) \leq (G, w)\). Then \((H, z) \in \text{Cover}(v)\), thus \( H = G \) from the minimality of \((G, w)\). Reversely, let \( G \in \text{Cause}_{u,F}(v) \) for some \((u, F) \in \mathcal{F}\). Then there exists \( w \) such that \((G, w)\) minimal in \( \text{Cover}_{u,F}(v) \).

To prove that \((G, w)\) is also minimal in \( \text{Cover}(v) \), consider \((H, k) \leq (G, w)\) in \( \text{Cover}(v) \). Since \( k \leq w \leq u - \text{lhs}(F) \) we get \((H, k) \in \text{Cover}_{u,F}(v)\), and so \( H = G \).
4 Quantitative Results

Theorem 10. Consider \( v \) obtainable from \( u \) through \( F \) and \( G \leq F \). Then \( G \in \text{Cause}_{u,F}(v) \) if and only if the following conditions hold:
- \( v \text{rhs}(G) \leq u - \text{lhs}(F) \);
- \( \text{rhs}(G) \cap v > \text{rhs}(G - r) \cap v \) for any \( r \in G \).

Consequently, \( G \in \text{Cause}(v) \) if and only if the following conditions hold:
- \( v \text{rhs}(G) \) stable;
- \( \text{rhs}(G) \cap v > \text{rhs}(G - r) \cap v \) for any \( r \in G \).

Proof. Consider \( G \in \text{Cause}_{u,F}(v) \). Then \( (G, v \text{rhs}(G)) \in \text{Cover}_{u,F}(v) \), and so the first condition holds. Suppose the second condition does not hold. Since \( \text{rhs}(G) \cap v \geq \text{rhs}(G - r) \cap v \) holds for any \( G, r \), we obtain that there exists a rule \( s \in G \) such that \( \text{rhs}(G) \cap v = \text{rhs}(G - s) \cap v \). Let \( G' = G - s \), then \( v \text{rhs}(G') = v \text{rhs}(G) \). We observe that \( (G', v \text{rhs}(G')) \) is also a cover for \( v \) with respect to \( u \) and \( F \). Since \( G' < G \), this contradicts the minimality of \( (G, v \text{rhs}(G)) \).

Now consider \( G \) such that the two conditions hold. We prove that \( (G, v \text{rhs}(G)) \) is a minimal cover. Clearly, \( (G, v \text{rhs}(G)) \) is a cover. Suppose that there is some other cover \( (G', w') \leq (G, v \text{rhs}(G)) \). Since \( v - w' \leq \text{rhs}(G') \), it follows that \( v \text{rhs}(G') \leq w' \). If \( G' < G \), then there exists \( s \in G \) such that \( G' \leq G - s \). Using the second condition, we obtain \( \text{rhs}(G') \cap v < \text{rhs}(G) \cap v \), and so \( v \text{rhs}(G) < v \text{rhs}(G') \leq w' \). This contradicts \( w' \leq v \text{rhs}(G) \). Therefore \( G' = G \).

If \( G \in \text{Cause}(v) \), then \( G \in \text{Cause}_{u,F}(v) \) for some \( u, F \); thus the conclusion. Reversely, if \( v \text{rhs}(G) \) stable, we have \( (G, v \text{rhs}(G)) \in \text{Cover}_{u,F}(v) \) for \( u = \text{lhs}(G) + v \text{rhs}(G), F = G \) and \( G \) fulfills the two characterization conditions. Hence \( G \in \text{Cause}_{u,F}(v) \subset \text{Cause}(v) \).

Note that the second condition can be rewritten as \( \text{rhs}(r) \cap (v \text{rhs}(G - r)) > 0 \) for all \( r \in G \) (using Proposition 2).

These results allow us to obtain the causes of a multiset directly, without looking for the set of covers. More precisely, they allow an inductive construction for the set of causes.

Consider a multiset \( v \) of objects. Let \( \mathcal{C}(v) \) be the smallest set with the properties:
- \( (0, v) \in \mathcal{C}(v) \);
- \( (G, w) \in \mathcal{C}(v) \) and \( r \) such that \( \text{rhs}(r) \cap w > 0 \), then \( (G + r, w \text{rhs}(r)) \in \mathcal{C}(v) \).

It is easy to see that \( (G, w) \in \mathcal{C}(v) \) implies \( w = v \text{rhs}(G) \). Now let \( \mathcal{C}'(v) \) be the set of elements \( (G, w) \) of \( \mathcal{C}(v) \) for which \( w \) is stable.

Theorem 11.

\[
\text{Cause}(v) = \{ G \mid \exists w \text{ such that } (G, w) \text{ minimal element of } \mathcal{C}'(v) \}.
\]
Proof. Let \( G \in \text{Cause}(v) \). We prove by induction on the size of \( H \) that for all \( H \leq G \) we have \((H, v \setminus \text{rhs}(H)) \in C(v)\). For \( H = 0 \) we have \((0, v) \in C(v)\). Let \( H' = H + r \leq G \). Since \( r \in G \) we have \( \text{rhs}(r) \cap (v \setminus \text{rhs}(G - r)) > 0 \) from Theorem 10. Using operation properties (Proposition 2) together with \( H \leq G - r \) we obtain \( \text{rhs}(r) \cap (v \setminus \text{rhs}(H)) > 0 \), and so \((H', v \setminus \text{rhs}(H')) = (H + r, (v \setminus \text{rhs}(H)) \setminus \text{rhs}(r)) \in C(v)\); this proves the inductive hypothesis. Hence \((G, v \setminus \text{rhs}(G)) \in C(v)\). It is also in \( C(v) \) since \( v \setminus \text{rhs}(G) \) is stable. To prove minimality, it is sufficient to note that any element of \( C(v) \) is also a cover for \( v \).

Reversely, consider \( G \) for which there exists a stable \( w \) such that \((G, w)\) minimal in \( C(v) \); then \( w = v \setminus \text{rhs}(G) \) is stable. Suppose there exists a rule \( s \) such that \( \text{rhs}(G) \cap v = \text{rhs}(G - s) \cap v \) and let \( H \) denote \( G - s \). Then \( v \setminus \text{rhs}(H) = v \setminus \text{rhs}(G) \) is stable and \((H, v \setminus \text{rhs}(H)) \leq (G, v \setminus \text{rhs}(G))\). Let \( r_1, \ldots, r_n \) be rules (not necessarily distinct) such that \( G = r_1 + \ldots + r_n \) and \( \text{rhs}(r_1) \cap v, \text{rhs}(r_2) \cap (v \setminus \text{rhs}(r_1)) \ldots, (\text{rhs}(r_n) \cap (v \setminus \text{rhs}(r_1) \ldots + r_{n-1})) > 0 \). Such a decomposition exists for \( G \) from the inductive construction of \( C(v) \). We let \( G_i \) denote \( r_1 + \ldots + r_i \).

There exists \( k \) such that \( s = r_k \). Let \( H_i \) denote \( G_i - r_k \), for \( k + 1 \leq i \leq n \). We have \((G_{k-1}, v \setminus \text{rhs}(G_{k-1})) \in C(v) \) and \( \text{rhs}(r_{i+1}) \cap (v \setminus \text{rhs}(G_{k-1})) \geq \text{rhs}(r_{i+1}) \cap (v \setminus \text{rhs}(G_{k-1})) > 0 \). Thus \((H_{k+1}, v \setminus \text{rhs}(H_{k+1})) \in C(v)\). Continuing this line of reasoning we obtain that, for \( H_n = H \), we have \((H, v \setminus \text{rhs}(H)) \in C(v) \) and thus in \( C(v) \) which contradicts the minimality of \((G, v \setminus \text{rhs}(G))\).

**Example 12.** Consider the membrane system of Example 5 with alphabet \( O = \{x, y, a, b\} \) and rules \( r_1 : x \rightarrow a + b \), \( r_2 : y \rightarrow b \). Let \( v = 2a \). Then \( C(v) = \{(0, 2a), (r_1, a), (2r_1, 0)\} \). \( a \) and \( 2a \) are stable thus \( \text{Cause}(v) = \{0, r_1, 2r_1\} \). Let \( v' = a + b \). Then \( C(v') = \tilde{C}(v') = \{(0, a + b), (r_1, 0), (r_2, a), (r_1 + r_2, 0)\} \) thus \( \text{Cause}(v') = \{0, r_1, r_2\} \). We see that \( r_1 + r_2 \) does not come from a minimal pair and is excluded from the set of causes; indeed, when \( r_1 \) is applied, it produces both \( a \) and \( b \), so \( r_2 \) is not needed to produce the \( b \) from \( a + b \).

To see how the stability requirement is involved in finding the causes of a multiset, consider the previous membrane system to which we add a third rule \( r_3 : a + b \rightarrow y \). Then, for \( v' = a + b \), we have \( C(v') = \{(0, a + b), (r_1, 0), (r_2, a), (r_1 + r_2, 0)\} \) but \( \tilde{C}(v') = \{(r_1, 0), (r_2, a), (r_1 + r_2, 0)\} \). This leads to \( \text{Cause}(v') = \{r_1, r_2\} \). We see that 0 is no longer a cause for \( v' \) since having it as a cause would mean that \( v' \) can be entirely obtained from objects \( a, b \) which are not evolved in an evolution step. This does not take place since if \( a \) and \( b \) were both in a multiset \( u \), when \( u \) evolved \( a + b \) would be consumed by rule \( r_3 \).

The characterization provided by Theorem 10 allows us to present a result concerning the way that local causes of resources produced in the same evolution step interact. We prove that common local causes of two multisets are also local causes of their reunion. However, this result cannot be extended at a global level since we cannot express globally the fact that \( x \) and \( y \) should be obtained from the same source \( u \). More precisely, if \( x \) and \( y \) have a common global cause \( G \), it does not follow that \( G \) is a global cause for \( x \cup y \) (Example 14).
Proposition 13. Consider $x, y$ obtainable from $u$ through $F$. If $G \in \text{Cause}_{u,F}(x)$ and $y \backslash \text{rhs}(G) \leq u - \text{lhs}(F)$, then $G \in \text{Cause}_{u,F}(x \cup y)$. As a consequence, $\text{Cause}_{u,F}(x) \cap \text{Cause}_{u,F}(y) \subseteq \text{Cause}_{u,F}(x \cup y)$.

Proof. We prove that $G$ fulfills the conditions of Theorem 10.

First, $(x \cup y) \backslash \text{rhs}(G) = (x \backslash \text{rhs}(G)) \cup (y \backslash \text{rhs}(G)) \leq u - \text{lhs}(F)$.

Second, suppose there exists a rule $s \in G$ such that $\text{rhs}(G) \cap (x \cup y) = \text{rhs}(G - s) \cap (x \cup y)$. Let $g_x$ and $g_y$ denote $\text{rhs}(G) \cap x$ and $\text{rhs}(G) \cap y$, while $g'_x$ and $g'_y$ denote $\text{rhs}(G - s) \cap x$ and $\text{rhs}(G - s) \cap y$, respectively. The previous equality translates to $g_x \cup g_y = g'_x \cup g'_y$, which we prove that leads to a contradiction.

Since $G \in \text{Cause}_{u,F}(x)$, we know that $g_x > g'_x$, thus there exists $a$ such that $g_x(a) > g'_x(a)$. We have two options for the values of $g_x$ and $g_y$ in $a$: either $g_x(a) \geq g_y(a)$ or $g_x(a) < g_y(a)$.

First option: if $g_x(a) \geq g_y(a)$ then $g'_x \cup g'_y(a) = g_x \cup g_y(a) = g_x(a)$. Since $g_x(a) \neq g'_x(a)$, it follows that $g_x(a) = g'_y(a)$ and that $g'_y(a) \leq g'_x(a)$, leading to a contradiction.

Second option: if $g_x(a) < g_y(a)$ we obtain $g'_x \cup g'_y(a) = g_y(a)$. Note that $g_x(a) > g'_x(a)$ implies $\text{rhs}(G - s)(a) < x(a)$ and $\text{rhs}(s)(a) > 0$ (otherwise we would have $g_x(a) = g'_x(a)$).

First case: Suppose $g'_x(a) < g'_y(a)$; then we obtain $g'_y(a) = g_y(a)$. Suppose $\text{rhs}(G)(a) < y(a)$; then we obtain $\text{rhs}(s)(a) = 0$ which contradicts $g_x(a) > g'_x(a)$. Thus $\text{rhs}(G)(a) > y(a)$ and so we also obtain $\text{rhs}(G - s)(a) \geq y(a)$, which leads to $x(a) > y(a)$. Returning to $g'_x(a) < g'_y(a)$, it implies that $\text{rhs}(G - s) \cap x(a) < y(a)$, which leads to a contradiction.

Second case: Suppose $g'_x(a) \geq g'_y(a)$; then we obtain $g'_y(a) = g_y(a)$. However, $g'_x(a) = \text{rhs}(G - s)(a) < \text{rhs}(G)(a)$ and $g_y(a) = (\text{rhs}(G) \cap y)(a)$. We obtain $g_y(a) = y(a) = \text{rhs}(G - s)(a)$. Recall that we are under the assumption that $g_x(a) < g_y(a)$, which can be rewritten to $\text{rhs}(G) \cap x(a) < y(a)$. Since $y(a) = \text{rhs}(G - s)(a)$, it means that $y(a) < x(a)$ and $y(a) > \text{rhs}(G)(a)$, which contradicts $g_x(a) < g_y(a)$.

Example 14. Consider a membrane system with alphabet $O = \{a, b, c, d\}$ and rules $r_1 : d \to c$, $r_2 : a + b \to d$. Let $x = a + c$ and $y = b + c$. Then $G = r_1 \in \text{Cause}(x) \cap \text{Cause}(y)$ but $G \notin \text{Cause}(x \cup y) = \emptyset$. Note that there is no $u$ such that $x$ and $y$ can be both obtained from it, which leads to $\text{Cause}(x \cup y)$ being empty.

Theorem 15. Let $u, v$ be multisets of objects. Then $v$ can be obtained from $u$ through $F$ if and only if there exist $G \in \text{Cause}(v)$, $H$ a multiset of rules and $w$ a stable multiset of objects such that $v \geq w \backslash \text{rhs}(G)$ and $u = \text{lhs}(G) + \text{lhs}(H) + w$.

Proof. If there exists $F$ such that $v$ can be obtained from $u$ through $F$ then $\text{Cause}_{u,F}(v)$ is not empty, thus $\exists G \in \text{Cause}_{u,F}(v) \subseteq \text{Cause}(v)$. Let $H = F - G$ and $w = u - \text{lhs}(F)$; they fulfill the requested conditions. Reversely, let $F = G + H$; since $w$ is stable, $F$ is maximally parallel with respect to $u$. Moreover, $u - \text{lhs}(F) + \text{rhs}(F) \geq w + \text{rhs}(G) \geq v$. 

This result provides a static causal characterization for the multisets of objects \( u \) from which \( v \) can be obtained, without actually performing all the possible evolution steps starting from \( u \).

**Example 16.** Consider the membrane system of Example 5 with alphabet \( O = \{ x, y, a, b \} \) and rules \( r_1 : x \rightarrow a + b, r_2 : y ightarrow b \). Let \( v = 2a \). We previously saw that \( \text{Cause}(v) = \{0, r_1, 2r_1\} \). Theorem 15 states that \( 2a \) can be obtained from \( u \) if and only if either \( u = \text{lhs}(H) + w \) for stable \( w \geq 2a \), or \( u = \text{lhs}(r_1) + \text{lhs}(H) + w \) for stable \( w \geq a \), or \( u = \text{lhs}(2r_1) + \text{lhs}(H) + w \) for any \( w \) stable. This can be rewritten as either \( u(a) \geq 2 \) or \( u(x) \geq 1 \), \( u(a) \geq 1 \) or \( u(x) \geq 2 \).

**Example 17.** For a more complicated example, consider the system with \( O = \{ x, y, a, b \} \) and rules \( r_1 : x \rightarrow a + b, r_2 : y ightarrow b, r_3 : a + b \rightarrow y \). Let \( v = a + b \). We saw in the second part of Example 12 that \( \text{Cause}(v) = \{r_1, r_2\} \). We apply Theorem 15, for \( G = r_1 \). Let \( H(r_1) = \alpha, H(r_2) = \beta \) and \( H(r_3) = \gamma \). Since \( w \) is stable and larger than \( v \backslash \text{rhs}(r_1) = 0 \) it follows that \( w \) can be of form \( \delta \cdot a \) or \( \omega \cdot b \), with \( \delta, \omega \geq 0 \). We obtain that \( u = x + \alpha \cdot x + \beta \cdot y + \gamma \cdot (a + b) + \delta \cdot a + \omega \cdot b \), thus \( u \) is characterized by \( u(x) \geq 1 \). If we consider \( G = r_2 \) then \( w \) is stable and \( w \geq a \) so \( w = \delta \cdot a \), with \( \delta \geq 1 \). With similar notations, we obtain \( u = y + \alpha \cdot x + \beta \cdot y + \gamma \cdot (a + b) + \delta \cdot a \), which translates to \( u(y) \geq 1 \) and \( u(a) \geq u(b) + 1 \).

Finally, we can state that \( a + b \) can be obtained from \( u \) if and only if either \( u(x) \geq 1 \) or \( u(y) \geq 1 \), \( u(a) \geq u(b) + 1 \).

## 5 Conclusion and Related Work

In this paper we have defined causality at both local and global level; local causality depends on a certain evolution step, while global causality takes into consideration all possible evolution steps. For local causality we proved that a common local cause for two multisets is also a local cause for their union. Similar properties for the other multiset operations remain to be found (at both local and global level). Several technical results concern the correspondence between local and global causality, as well as equivalent representations for causes. The global definition leads to an inductive procedure for finding the causes of a multiset (Theorem 11). The main result of the paper is a characterization in terms of causes for multisets \( u \) from which a fixed multiset \( v \) can be obtained (Theorem 15).

An earlier attempt to define a notion of causality over objects in membrane systems was made in [1]. The definition was given in terms of relating multisets of objects of form \( k \cdot a \). Moreover, the multiplicity \( k \) of \( a \) could only be taken to be \( u(a) \) or \( u'(a) \) from an evolution step \( u \overset{r}{\rightarrow} u' \). The definition in this paper is more general, allowing to relate (locally) any multiset \( v \leq u \) with a multiset of rules \( G \), and implicitly with a multiset of objects \( \text{lhs}(G) + v \backslash \text{rhs}(G) \leq u \).

In paper [4], rules and objects (not multisets of objects) are causally related one by one, with respect to the order of rule application, without offering a formal approach. Instead, the paper focuses on giving a semantics for the membrane system in which causal information is defined by the labels of a transition system.
A formal approach for a causal semantics was introduced in [5], where the authors introduce an event structure semantics based on sequential rule application, without a clear link to multisets of objects.

Paper [7] discusses causality in terms of two Petri net representations of a membrane system. The notion of causality is also based only on rule applications and is obtained by unfolding each Petri net representation. In the same research line, paper [10] proposes an event based view of membrane evolution by translating membrane systems into zero safe nets and associating an event automaton to the resulting nets.

All these papers propose a notion of causality based on the temporal order of single rule application. Such approaches cannot tackle a quantitative view of causality like we introduce in this paper, which offers solutions of quantitative problems such as the characterization from Theorem 15.

A related problem to that solved by Theorem 15 is presented in [2], where we investigated how to find all the multisets \( u \) which evolve to a given multiset \( u' \) in a single evolution step. If \( u \) evolves to \( u' \), then a necessary condition is that \( v = u' \) is obtainable from \( u \), thus \( u \) is characterized by the causes of \( v \) in terms of Theorem 15. Note that we solved the \( u, u' \) problem in a dynamic manner, by reversing the evolution of the P system in question. The results of this paper suggest a static solution of the problem.

Acknowledgements

We thank the anonymous reviewers for their helpful comments.

References

Circular Post Machines and P Systems with Exo-Insertion and Deletion

Artiom Alhazov\textsuperscript{1,3}, Alexander Krassovitskiy\textsuperscript{2}, and Yurii Rogozhin\textsuperscript{1}

\textsuperscript{1} Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău MD-2028 Moldova
E-mail: \{artiom,rogozhin\}@math.md

\textsuperscript{2} Research Group on Mathematical Linguistics
Rovira i Virgili University
Av. Catalunya 35, Tarragona 43002 Spain
E-mail: alexander.krassovitskiy@estudiants.urv.cat

\textsuperscript{3} Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
E-mail: artiom.alhazov@unimib.it

Abstract. This paper focuses on P systems with one-symbol insertion and deletion without contexts. The main aim of this paper is to consider the operations applied at the ends of the string, and prove the computational completeness in case of priority of deletion over insertion. This result presents interest since the strings are controlled by a tree structure only, and because insertion and deletion of one symbol are the simplest string operations.

To obtain a simple proof, we introduce here a new variant (CPM5) of circular Post machines (Turing machines moving one-way on a circular tape): those with instructions changing a state and either reading one symbol or writing one symbol. We believe CPM5 deserves attention as a simple, yet useful tool.

In the last part of the paper, we return to the case without priorities. We give a lower bound on the power of such systems, which holds even for one-sided operations only.

1 Introduction

Insertion and deletion are fundamental string operations, introduced in the formal language theory mainly with linguistic motivation. The biological motivation is that these operations correspond to mismatched annealing of DNA sequences. They are also present in the evolution processes as point mutations as well as in RNA editing, see [4], [28] and [25].

In general, insertion/deletion means adding/removing a substring to/from a given string in a specified (left and right) context. A language-generating device can be defined by a finite set of insertion-deletion rules together with a set
of axioms: the language is obtained by iterating the operations, as the set of terminal strings from the closure of the axioms under the operations.

We mention some sources concerning the topic of this paper; see [8] for insertion systems, [19] for the idea of context adjoining, [12] for insertion and deletion. Related formal language theoretic research can be found in e.g., [10], [21] and [23], and the biologically motivated studies can be found in e.g., [5], [11], [25] and [29].

As expected, insertion-deletion systems characterize recursively enumerable languages, which is not surprising since the systems the context dependency and the erasing ability. However, the contexts may be replaced by insertion and deletion of strings of sufficient length, in a context-free manner, [20]. If the length is not sufficient (at most two symbols), then such systems are decidable and their characterization was shown in [30].

Similar research continued in [22] and [16] on insertion-deletion systems with one-sided contexts, i.e., where the context dependency is present only from the left (right) side of all insertion and deletion rules. The behavior of such variants is between those of context-free and context-dependent insertion-deletion systems. Indeed, like in the context-free systems, an insertion or deletion may be performed any number of times, but like in the contextual variant, their application site is not arbitrary. The above papers give several computational completeness results depending on the size of parameters of insertion or deletion rules, i.e., the bounds on the lengths of strings in the description of rules, \((\text{inserted string}, \text{its left context, its right context}; \text{deleted string}, \text{its left context, its right context})\); some combinations do not lead to computational completeness and there are languages that cannot be generated by such devices.

In [14] one-sided insertion-deletion systems with insertion and deletion rules of at most two symbols were considered. This corresponds to systems of size \((1, 1, 0; 1, 1, 0), (1, 1, 0; 1, 0, 1), (1, 1, 0; 2, 0, 0)\) and \((2, 0, 0; 1, 1, 0)\), where the first three numbers represent the maximal length of the inserted string and the maximal length of the left and right contexts, while the last three numbers represent the same information for deletion rules. A characterization in terms of context-free grammars of the class of insertion-deletion systems of size \((1, 1, 0; 1, 1, 0)\) was presented. It was also shown that such systems generate some non-regular context-free languages even without deletion. The remaining classes are not computationally complete; the language \((ba)^+\) cannot be generated by such systems.

In [14] one also considered insertion-deletion operations in P systems framework, which is a maximally parallel distributed computing device inspired by the structure and the functioning of a living cell. We refer to [24], [26], [7] and [31] for more details about P systems. It was shown that such additional control permits to increase the computational power up to computationally completeness results for all four cases, improving the results from [17] and [15]. However, the framework of P systems cannot increase the computational power to such extent in all cases, namely it was shown that if context-free insertion and deletion rules using at most two symbols are considered, i.e. systems of size \((2, 0, 0; 2, 0, 0)\), then the corresponding P systems are still not computationally complete [14,
It is thus interesting to consider conditions that would allow such systems to reach computational completeness.

In [1] one considers insertion-deletion P systems with insertion and deletion operations applied at the ends of the string (called exo-operations). Such systems with insertion of one symbol and deletion of up to two symbols and systems with insertion of up to two symbols and deletion of one symbol are computationally complete. The question about the computational power of insertion-deletion P systems with one-symbol insertion and one-symbol deletion at the ends of string has been left open (except the computational completeness holds in tissue case).

We should also mention the research in [9]: one considers insertion at one end of a string coupled with deletion at the other end. Even when the pairing is not prescribed, the universality is still achieved. However, the size of the inserted and deleted substrings is not bounded.

In this paper we deal with one-symbol exo-insertion and one-symbol exo-deletion with priority of the latter. We also introduce a new variant (CPM5) of circular Post machines and use it for constructing a simpler proof; we believe CPM5 presents interest in itself, as a convenient tool for similar proofs. Finally, we give a lower bound on the power of systems without priorities, by a construction with one-sided operations.

2 Preliminaries

We recall some notions we shall use throughout the paper. An alphabet is a non-empty finite set of symbols. The cardinality of a finite set \( A \) is written as \( \text{card}(A) \). A sequence of symbols from an alphabet \( V \) is called a word over \( V \). The set of all words over \( V \) is denoted by \( V^* \), and the empty word is denoted by \( \lambda \); we use \( V^+ = V^* \setminus \{\lambda\} \). The length of a word \( x \) is denoted by \( |x| \), while we denote the number of occurrences of a symbol \( a \) in a word \( x \) by \( |x|_a \). For each non-empty word \( x \), \( \text{alph}(x) \) is the minimal alphabet \( W \) such that \( x \in W^* \).

We do not present here definitions concerning standard concepts of the theory of formal languages and we refer the reader to [27] for more details. We denote the families of regular and recursively enumerable languages by \( \text{REG} \) and \( \text{RE} \), respectively.

2.1 Circular Post machines

The following model (CPMs) has been introduced in [18], where it was shown that all introduced variants of CPMs (CPM0-CPM4, see Table 1) are computationally complete; we first recall the deterministic CPM0s (Circular Post machines of type 0).

**Definition 1.** A circular Post machine (of type 0) is a tuple \((\Sigma, Q, q_1, q_f, R)\) with a finite alphabet \( \Sigma \) where \( 0 \in \Sigma \) is the blank, a finite set of states \( Q \), the initial state \( q_1 \in Q \), the final state \( q_f \in Q \), and a finite set of instructions \( R \) with all instructions having one of the forms \( px \to q \) (erasing the symbol read by
Table 1. Variants of circular Post machines

<table>
<thead>
<tr>
<th>CPM0</th>
<th>CPM1</th>
<th>CPM2</th>
<th>CPM3</th>
<th>CPM4</th>
<th>CPM5</th>
</tr>
</thead>
<tbody>
<tr>
<td>px → q</td>
<td>px → q</td>
<td>px → q</td>
<td>px → q</td>
<td>px → q</td>
<td>px → q</td>
</tr>
<tr>
<td>px → y</td>
<td>px → y</td>
<td>px → y</td>
<td>px → y</td>
<td>px → y</td>
<td>px → y</td>
</tr>
<tr>
<td>p0 → yq</td>
<td>p0 → yq</td>
<td>p0 → yq</td>
<td>p0 → yq</td>
<td>p0 → yq</td>
<td>p0 → yq</td>
</tr>
</tbody>
</table>

We also refer to all instructions with qf in the right hand side as halt instructions.

Notice that a circular tape can be thought of as a finite string of symbols (from the one following the state to the one preceding the state in the circular representation). In this way, CPM0 is a finite-state machine, which reads the leftmost symbol of the string, possibly consuming it, and uses the symbol-state information to change the state, possibly writing a symbol on the right.

Table 1 summarizes the difference between the variants of CPMs, [18] (we also include CPM5 defined in subsection 3.1). The difference between CPMk, 0 ≤ k ≤ 4, is only in the way the lengthening instruction works: whether it applies to any symbol or only to the blank, whether it introduces one of the two new symbols after the state, and whether one of the new symbols equals the one read. Although all variants are computationally equivalent, these distinctions can affect the size of the smallest universal machines.

2.2 Insertion and deletion

Insertion and deletion rules (grouped in sets I and D, respectively) are triples of the form (u, α, v), α ≠ λ, where u, α, v are strings over V, the alphabet of the system. We denote by ⇒ins the relation defined by an insertion rule; formally, x ⇒ins y iff for some (u, α, v) ∈ I and x1, x2 ∈ V*

\[ x = x_1uαv, \quad y = x_1uαv \] \tag{1}

and by ⇒del the relation defined by a deletion rule; formally, x ⇒del y iff for some (u, α, v) ∈ D and x1, x2 ∈ V*

\[ x = x_1uvα, \quad y = x_1uvα \] \tag{2}

We say that insertion or deletion is applied at the left (right) end of the string if x1 = λ (x2 = λ) in (1, 2). In particular, for the rules without context, the operations can be written as

\[ x \Rightarrow_{ins, (α)} αx, \quad x \Rightarrow_{ins, (α)} xα, \quad αx \Rightarrow_{del, (α)} x, \quad xα \Rightarrow_{del, (α)} x. \]
2.3 Distributed insertion-deletion

Definition 2. (See also [24].) An insertion-deletion P system (IDP) of degree \( n \) is the following construct:

\[
\Pi = (V, T, \mu, M_1, \ldots, M_n, R_1, \ldots, R_n),
\]
where
- \( V \) is a finite set called the alphabet,
- \( T \subseteq V \) is the terminal alphabet,
- \( \mu \) is the membrane (tree) structure of the system of \( n \) regions (nodes). It is represented by a word containing correctly nested marked parentheses. Note: we use notation \( \prod_{i \in I} (\mu_i) \) to denote concatenation of strings \( \mu_i \) for all \( i \in I \); the tree does not depend on the order of concatenation.
- \( M_i \), for each \( 1 \leq i \leq n \), is a finite language associated to the membrane \( i \).
- \( R_i \), for each \( 1 \leq i \leq n \), is a set of insertion and deletion rules with target indicators associated to membrane \( i \) and which have the following form:
\[
(u, x, v; \text{tar})_a \in R_i, \quad \text{where} \quad (u, x, v) \text{ is an insertion rule, and } (u, x, v; \text{tar})_a \in R_i, \quad \text{where} \quad (u, x, v) \text{ is a deletion rule, and } \text{tar}, \text{ called the target indicator, is from the set } \{\text{here}, \text{in}_j, \text{out} | 1 \leq j \leq n\}.
\]

In case of tissue systems, a graph is given instead of a tree, the environment is a distinguished node, other nodes are called cells, and targets are the node labels.

A P system (of degree \( n \)) with insertion and deletion exo-operations (eIDP) is an insertion-deletion P system (of degree \( n \)) with insertion and deletion rules applied at the ends of string and with target indicators associated to membrane. We denote by eIDtP a tissue P system with insertion-deletion exo-operations. In this paper we use insertion and deletion exo-operations in context-free manner, so we can write these operations as follows: \( \text{ins}_a(x, \text{tar}) \) and \( \text{del}_a(x, \text{tar}) \), where \( \text{ins}_a(x, \text{tar}) \) is an insertion rule applying at the left (right) end of string, and \( \text{del}_a(x, \text{tar}) \) is a deletion rule applying at the left (right) end of string (if \( \alpha = \ell \) or \( \alpha = r \) accordingly). For example, if rule \( \text{ins}_l(x, \text{out}) \) is applied to string \( w \) then the string \( xw \) will be sent to the outer region.

Let us illustrate the difference between IDPs and eIDPs by a simple example. A system \( \Pi = (\{a, b\}, \{a, b\}, [1], \{babbab\}, \{\text{del}(b), \text{ins}(a)\}) \) generates language \( L(\Pi) = \{w \in \{a, b\}^* | |w|_b \leq 4\} \). However, a system \( \Pi' = (\{a, b\}, \{a, b\}, [1], \{babbab\}, \{\text{del}(b), \text{ins}(a)\}) \) generates language \( L(\Pi') = (\{\lambda\} \cup \{b\})\{abbab\}\{a\}^* \).

Any \( n \)-tuple \((N_1, \ldots, N_n)\) of languages over \( V \) is called a configuration of \( \Pi \). For two configurations \((N_1, \ldots, N_n)\) and \((N'_1, \ldots, N'_n)\) of \( \Pi \) we write \((N_1, \ldots, N_n) \Rightarrow (N'_1, \ldots, N'_n)\) if one can pass from \((N_1, \ldots, N_n)\) to \((N'_1, \ldots, N'_n)\) by applying the insertion and deletion rules from each region of \( \mu \), in maximally parallel way, i.e., in parallel to all possible strings from the corresponding regions, and following the target indications associated with the rules. We assume that every string represented in the region has arbitrary many copies. Hence, all rules applicable to a string are applied in the same step, the original string remains, and the multiplicities are not tracked.

More specifically, if \( w \in N_i \) and \( r = (u, x, v; \text{tar})_a \in R_i \), respectively \( r = (u, x, v; \text{tar})_a \in R_i \), such that \( w \Rightarrow_{\text{ins}} w' \), respectively \( w \Rightarrow_{\text{del}} w' \), then \( w' \)
will go to the region indicated by tar. If tar = here, then the string remains in \( N_i \), if tar = out, then the string is moved to the region immediately outside the membrane \( i \) (maybe, in this way the string leaves the system), if tar = in, \( j = 1, \ldots, n \), then the string is moved to the immediately below \( j \)-th region.

A sequence of transitions between configurations of a given insertion-deletion P system \( \Pi \), starting from the initial configuration \( (M_1, \ldots, M_n) \), is called a computation with respect to \( \Pi \). The result of a computation consists of all strings over \( T \) which are sent out of the system or are sent to special “output” region at any time during the computation. We denote by \( L(\Pi) \) the language of all strings of this type. We say that \( L(\Pi) \) is generated by \( \Pi \).

As in [24] we denote by \( ELSP_k(ins_{m,m'}, del_{n,n'}) \) the family of languages generated by insertion-deletion P systems of degree at most \( k \geq 1 \) having the size \( (m, m', m''); (n, n', n'') \). We denote by \( SP_k(e^{-ins_{m,m'}}, e^{-del_{n,n'}}) \) the class of P systems of degree at most \( k \geq 1 \) with insertion and deletion exo-operations having the size \( (m, m'; n, n') \) and by \( StP_k(e^{-ins_{m,m'}}, e^{-del_{n,n'}}) \) the class of tissue P systems of degree at most \( k \geq 1 \) with insertion and deletion exo-operations having the size \( (m, m''; n, n'') \). If we only consider insertion/deletion at the right (left) end, we replace \( e^{-ins}/e^{-del} \) with \( r^{-ins}/l^{-del} \) in the notation, respectively.

We also consider insertion-deletion P systems with insertion and deletion exo-operations where deletion rules have a priority over insertion rules; the corresponding class of systems we denote as \( SP_k(e^{-ins_{m,m'}}, e^{-del_{n,n'}}) \).

3 Results

It has been shown in [18] that any Turing machine can be simulated by a CPM0. To obtain, e.g., a language generating device, it suffices to introduce non-determinism and perform minor post-processing (such as removing the blanks at the ends of the halting configuration; doing this by a CPM is an easy exercise for the reader).

We now define another string-processing system that is computationally complete, is suitable for easy simulation by P systems with exo-insertion and exo-deletion, and has a low descriptional complexity of the rule types.

3.1 A new variant of Circular Post machines

We introduce a new variant of Circular Post Machines.

**Definition 3.** A CPM5 is a tuple \( (\Sigma, Q, q_1, q_f, R) \) with instructions of the following types (we use the notation \( Q' = Q \setminus \{q_f\} \)):

- \( px \rightarrow q, p \in Q', q \in Q, x \in \Sigma \), the same type as in CPM0. The corresponding computational step is \( pxW \xrightarrow{p \rightarrow x} qW, W \in \Sigma^* \).
- \( p \rightarrow yq, p \in Q', q \in Q, y \in \Sigma \). This is the new type of rule. Notice it does not consume a symbol. The corresponding computational step is \( pW \xrightarrow{p \rightarrow yq} qWy, W \in \Sigma^* \).
Theorem 1. Any CPM0 $P$ can be simulated by CPM5 $P'$.

Proof. Consider a CPM0 $P = (\Sigma, Q, q_1, q_f, R)$ with $0 \in \Sigma$, and $Q' = Q \setminus \{q_f\}$.
We construct CPM5 $P' = (\Sigma, Q', q_1, q_f, R)$ simulating of CPM0 $P$:

$$
\hat{\Sigma} = \Sigma \cup \{a_q \mid q \in \Sigma\},
\hat{Q} = Q \cup \{p_x \mid (px \to qy) \in Q\} \cup \{p_0, p'_0 \mid (p0 \to yq0) \in R\} \cup \{r', r\},
\hat{R} = \{px \to q \mid (px \to q) \in R\} \cup \{px \to p_x, px \to yq \mid (px \to yq) \in R\}
\cup \{p0 \to p_0, p_0 \to yp'_0, p'_0 \to a_q r', ra_q \to q \mid (p0 \to yq0) \in R\}
\cup \{r' \to 0r\} \cup \{rx \to rz, rz \to xr \mid x \in \Sigma\}.
$$

We claim $P'$ is equivalent to $P$ (it produces the same result). Let $pxW \xRightarrow{px \to qy} qWy$ be a computational step in $P$. The corresponding derivation in $P'$ is $pxW \xRightarrow{px \to qy} p_x W \xRightarrow{p_x \to qy} p_0 x W \xRightarrow{p_0 x W \to qy_0} p'_0 x W \xrightarrow{a_q r'} r' x W y a_q \xrightarrow{r' \to 0r} r x W y a_q 0$

Consider the second case. Let $p0 x W \xrightarrow{p0 \to yq0} q0 x W y$ be a computational step in $P$. The corresponding derivation in $P'$ is $p0 x W \xrightarrow{p0 \to q0} p'_0 x W \xrightarrow{p'_0 \to qy_0} p'_0 x W \xrightarrow{r' x W y a_q 0} r' x W y a_q 0 \xrightarrow{r' \to 0r} r x W y a_q 0$

It remains to notice that $P'$ does not do anything more than the simulation of $P$. Indeed, even in the non-deterministic case, within the simulation of some computation step of $P$, $P'$ is in some state $\hat{Q} \setminus Q$, whose behaviour is bound to such simulation by construction. \qed

Remark 1. If we do not require the simulation to go through all configuration of the computation, then a faster simulation of any CPM0 $P$ by a CPM5 $P'$ is possible. Instructions $px \to q$ are performed in one step, and instructions $px \to yq$ are performed in two steps, so it suffices to consider instructions $p0 \to yq0$.

Like in the proof above we may start with $p0 \to p_0$, $p_0 \to yp'_0$. Informally $p'_0$ corresponds to the state $q$, except symbol $0$ has not been introduced, i.e., to the situation where it has “already been read” in state $q$. Hence, depending on the next instruction of $P$ we sometimes replace $p'_0$ with an appropriate state, as follows. If $P$ has $q0 \to s$, then it suffices to set $p'_0 = s$. If $P$ has $q0 \to zs$, then it suffices to have a rule $p'_0 \to zs$. Finally, if $P$ has $q0 \to zs0$, then it suffices to set $p'_0 = q_0$, so the simulation of the first instruction continues by simulation of the next one without the first step.

Consider the non-deterministic case. In fact, the following restriction of non-determinism suffices:

Definition 4. An NCPM5 is a tuple $(\Sigma, Q, q_1, q_f, R)$, where

- $Q \setminus \{q_f\} = Q_1 \cup Q_2$, where $Q_1 \cap Q_2 = \emptyset$,
- For every $p \in Q_1$ and every $x \in \Sigma$, there is exactly one instruction of the form $px \to y$,
For every \( p \in Q_2 \), there are two instructions of the form \( p \rightarrow yq_1, p \rightarrow yq_2 \) (and the machine is deterministic if \( q_1 = q_2 \) for every pair of instructions \( p \rightarrow yq_1, p \rightarrow yq_2 \)).

**Corollary 1.** The class of (N)CPM5 is computationally complete.

**Proof.** The statement is a trivial consequence of the theorem above in the sense of completeness as simulating Turing machines or as computing partial recursive functions. Let us prove the claim of the corollary also in the sense of generating languages. We claim that NCPM5 generates all recursively enumerable languages. Indeed, for every \( L \in \text{RE} \) there exists a deterministic Turing machine \( M \) that, starting with configurations \( q_1 \rightarrow 1q_1, q_1 \rightarrow 1q_2 \), where the rest of the machine starts with \( q_2 \) and is a deterministic CPM5 that simulates a CPM0 simulating \( M \). □

### 3.2 P systems with priority of exo-deletion

We now proceed to the main result of the paper.

**Theorem 2.** Any NCPM5 \( P \) can be simulated by an eIDP \( II \) of size \((1, 0, 0; 1, 0, 0)\) with priority of deletion rules over insertion rules.

**Proof.** Consider an NCPM5 \( P = (\Sigma, Q, q_0, q_f, R) \) with symbols \( \Sigma = \{a_j \mid 0 \leq j \leq n\} \), where \( a_0 = 0 \) is the blank symbol, and states \( Q = \{q_i \mid 1 \leq i \leq f\} \), where \( q_1 \) is the initial state and the only terminal state is \( q_f \in Q \); let \( Q' = Q \setminus \{q_f\} \).

A configuration \( v = q_ia_jW \) of CPM5 \( P \) describes that \( P \) in state \( q_i \in Q \) considers symbol \( a_j \in \Sigma \) to the left of \( W \in \Sigma^* \). This configuration corresponds to the string \( v \) in the skin \( i_s \) of eIDP \( II \). The final configuration \( q_f a_jW \) of \( P \) corresponds to the string \( a_jW \) in the output membrane \( i_f \) of \( II \).

We now construct eIDP \( II = (V, \Sigma, \mu, M_{i_1}, \ldots, M_{i_j}, R_{i_1}, \ldots, R_{i_f}) \):

\[
V = \Sigma \cup Q \cup \{m_i, m_{\#}, z_i \mid q_i \in Q\} \cup \{n_i \mid q_i \in Q'\} \\
\cup \{n_{ij} \mid q_ia_j \rightarrow q_i \in R\} \cup \{S\},
\]

\[
\mu = \prod_{q_i \in Q} \left( \prod_{q_j \in Q'} \left( \prod_{m_i} \left( \prod_{n_i} \right) \right) \right),
\]

\[
\mu_i = \prod_{q_ia_j \rightarrow q_i \in R} \left( \prod_{n_{ij}} \right),
\]

\( M_{i_s} = \{v\} \) (the input),

\( M_i = \emptyset, i \neq i_s \), and the rules are given and explained below.

The membrane structure of \( II \) consists of the skin membrane \( i_s \), output membrane \( m_f \), inner membranes \( m_i, 1 \leq i \leq f \), “trap” membranes \( m_{\#}, \) inner membranes \( n_i, 1 \leq k \leq f \), depended on rules of \( P \) to be simulated; the parts of
the tree structure corresponding to $\mu$ relevant for explanation of the cases are also graphically shown in Figures 1 and 2 (we also indicate the associated rules).

\[
R_{i_s} = \{1_i : del_i(q_i, m_i) \mid q_i \in Q\} \cup \{2_i : del_i(S, here)\}, \ 3_i : del_i(z_f, i_f)\} \\
\cup \{4_i : del_i(z_i, n_i) \mid q_i \in Q\}, \\
R_{m_i} = \{1_i : del_i(q_j, m_{#i}) \mid q_j \in Q\} \cup \{2_i : ins_i(z_i, out)\}, \ q_i \in Q,
\]

NCPM5 $P$ starts a computation from a configuration $q_1a_jW$ and eIDP $\Pi$ starts computation from a string $q_1a_jW$ in the skin $i_s$ accordingly (other regions of $\Pi$ are empty). Further we continue to construct $\Pi$ and describe two cases how $\Pi$ simulates all types of rules of NCPM5 $P$.

(A) Rule $q_ia_j \rightarrow q_i \in R, q_i \in Q', q_j \in Q, a_j \in \Sigma$ is simulated as follows.

Let $q_ia_jW$ be a computation step in $P$. i.e., rule $q_ia_j \rightarrow q_i$ is applied to configuration $q_ia_jW$ yielding $q_iW$ ($W \in \Sigma^*$). The membrane substructure of $\Pi$ used for simulating this rule is illustrated in Figure 1.

\[
R_{i_s} = \{1_j : del_i(a_j, n_{ij})\} \cup \{2 : ins_i(S, out)\}, \ (q_ia_j \rightarrow q_i) \in R, \\
R_{m_{ij}} = \{1_i : ins_i(q_i, out)\}, \ (q_ia_j \rightarrow q_i) \in R,
\]

In $\Pi$ in region $i_s$ rule $1_i : del_i(q_i, m_i)$ is applied to string $q_ia_jW$, and the resulting string $a_jW$ moves to region $m_i$. We denote this action as follows: $q_ia_jW \rightarrow (i_s \downarrow j \downarrow m_i) a_jW$.

In region $m_i$ system $\Pi$ again checks the leftmost symbol of string: is it some $q_j \in Q$ or not. Recall that we consider systems with priority of deletion operations over insertion operations, and therefore rule $1_i : del_i(q_j, m_{#i})$ must be applied if it is possible. In this case the string enters the “trap” membrane $m_{#i}$ and will stay there forever. If it is not a case, then string $z_iW$ comes back to the skin $i_s$ by rule $2_i : ins_i(z_i, out)$.

Next, in the skin rule $4_i : del_i(z_i, n_i)$ is applied, and string $a_jW$ appears in region $n_i$. In region $n_i$ rule $1_j : del_i(a_j, n_{ij})$ must be applied, and string $W$
Fig. 2. The structure for an inserting rule

enters membrane $n_{ij}$, where rule 1 : $\text{ins}_l(q_l, \text{out})$ is applied and string $q_lW$ returns to region $n_i$. In this region only rule 2 : $\text{ins}_l(S, \text{out})$ may be applied, and the resulting string $S_qW$ appears in the skin $i_s$. In $i_s$, symbol $S$ will be removed by rule 2 : $\text{del}_l(S, \text{here})$, and string $q_lW$ is ready to continue the evolution.

We describe the evolution of string $q_lW$ from the skin as follows:

$$q_i = q_lW \mathbin{(i_s, 1)}_{m_i} a_j W \mathbin{(m_i, 2)}_{i_s} z_{i_s} a_j W \mathbin{(i_s, 1)}_{n_i} W \mathbin{(n_i, 1)}_{n_{ij}} W$$

Thus eDP $\Pi$ correctly simulates rule $q_lW$ of NCPM5 $P$.

(B) Rule $q_l \rightarrow a_kq_l$, $q_l \in Q', q_l \in Q, a_k \in \Sigma$ is simulated as follows.

Let $q_lW \mathbin{a \rightarrow a_k} W$ be a computation step in $P$, i.e., rule $q_l \rightarrow a_kq_l$ is applied to configuration $q_lW$ yielding $q_lW a_k$ ($W \in \Sigma^*$). A structure of $\Pi$ used in simulation of this rule of $P$ is presented in Figure 2.

$$R_{n_i} = \{1 : \text{ins}_l(q_l, \text{here}), 2 : \text{ins}_l(a_k, \text{out}) \mid q_l \rightarrow a_kq_l \in R\}.$$ 

The beginning of the solution of string $q_lW$ starting from the skin is the same as in Case (A):

$$q_i = q_lW \mathbin{(i_s, 1)}_{m_i} W \mathbin{(m_i, 2)}_{i_s} z_{i_s} W \mathbin{(i_s, 1)}_{n_i} W.$$ 

Notice that in region $n_i$ rule 1 : $\text{ins}_l(q_l, \text{here})$ must be applied first (otherwise string $W a_k$ comes to the skin by rule 2 : $\text{ins}_r(a_k, \text{out})$ and will stay there forever). Moreover, this rule 1 must be applied exactly once, after that rule 2 is applied and string $q_lW a_k$ moves to the skin. Now this string is ready to continue the evolution. We continue describing the evolution of the string from region $n_i$:

$$W \mathbin{(n_i, 1)}_{n_i} q_l W \mathbin{(n_i, 2)}_{i_s} q_l W a_k.$$ 

If rule 1 is applied more than once, string $q_lW W'$, $W' \in \{\Sigma \cup Q\}^*$ appears in the skin ($W'$ may be different from $l$ because $P'$ might apply two different rules in group 1 of $R_{n_i}$ if NCPM5 $P$ has two rules starting in $q_l$), and two symbols $q_lW$ at the left end of the string causes system $\Pi$ to send the string to the “trap” membrane $m_{\#i}$, and thus to block the computation:
Thus eIDP $Π$ correctly simulates rule $q_i \rightarrow akql$ of CPM5 $P$.

Notice that string $W$ can only appear in the output membrane if after applying rule 3 : $del(zf, if)$ in the skin, and string $zfW$ can appear in the skin only after correctly simulated derivation of NCPM5 $P$. Thus $Π$ correctly simulates $P$. \qed

**Corollary 2.** Notice also that the proof does not use deletion at the right.

$LSP^*(e - ins_1^{0,0} < e - del_1^{0,0}) = LSP^*(e - ins_1^{0,0} < l - del_1^{0,0}) = RE$.

The power of the systems with deletion only at the left and insertion only at the right is an interesting open question.

### 3.3 One-sided insertion/deletion without priorities

We emphasize the main open problem we try to attack:

\textbf{P:} Find a characterization of $P$ systems with exo-insertion of weight one and exo-deletion of weight one without contexts?

A number of related results is known (the references are given in the introduction, we repeat them here in a compact formulation for easy comparison):

- Not computationally complete if operations (even both with weight two) are performed anywhere in the string.
- Computationally complete if insertion has weight two.
- Computationally complete if deletion has weight two.
- Computationally complete for tissue $P$ systems.
- Computationally complete if deletion has priority over insertion (even without deletion on the right).

In this subsection we start show a lower bound, by a construction that only uses one-sided operations.

**Lemma 1.** Any deterministic finite automaton $P$ can be simulated by an eIDP $Π$ of size $(1,0,0;1,0,0)$ (without priorities) and with insertions and deletions at the right only.

**Proof.** Consider a deterministic finite automaton $P = (Σ, Q, q_1, F, σ)$ with symbols $Σ$, states $Q$ transition function $σ$, initial state $q_1$ and final states $F ⊆ Q$.

We now construct an eIDP $Π = (V, Σ, μ, M_{is}, ..., M_{if}, R_{is}, ..., R_{if})$:

$$V = Σ \cup Q \cup \{m_i \mid q_i ∈ Q\} \cup \{n_{ij} \mid σ(q_i, a_j) = q_l\} \cup \{S\},$$

$$μ = \prod_{q_i ∈ Q} \left( \prod_{σ(q_i, a_j) = q_l} [m_i \mid n_{ij} \mid m_j] \right) \mid i \neq is,$$

$$M_is = \{q_1\},$$

$$M_i = \emptyset, \ i ≠ is, \ and \ the \ rules \ are \ given \ and \ explained \ below.$$
The membrane structure of $\Pi$ consists of the skin membrane $i_s$, output membrane $i_f$, inner membranes $m_i$ and $n_{ij}$, where $q_i \in Q$, $\sigma(q_i, a_j) = q_l$.

$$R_{i_s} = \{1 : del_r(q_i, m_{i_l}) \mid q_i \in Q\} \cup \{2 : del_r(S, \text{here})\} \cup \{3_k : del_r(q_k, i_f) \mid q_k \in F\},$$

$$R_{m_i} = \{1_j : ins_r(a_j, n_{ij})\} \cup \{2 : ins_r(S, \text{out})\},$$

$$\sigma(q_i, a_j) = q_l,$$

$$R_{n_{ij}} = \{1 : ins_r(q_i, \text{out})\},$$

$$\sigma(q_i, a_j) = q_l.$$ 

We claim that $L(P) = L(\Pi)$: $\Pi$ generates exactly the language accepted by $P$. Indeed, eIDP $\Pi$ starts the computation from a string $q_l$ in the skin $i_s$ (other regions of $\Pi$ are empty).

A transition rule of $P \sigma(q_i, a_j) = q_l, q_i, q_l \in Q, a_j \in \Sigma$ is simulated as follows:

$$q_lW((i_1, i_2), m_{i_l})W((m_{i_l}, i_2), n_{ij})a_jW((n_{ij}, i_2), m_i)q_0a_jW((m_i, i_2), q_1)q_0a_jW.$$ At the end of computation, $q_k \in F$ is removed by rule $3_k : del_r(q_k, i_f)$ and the resulted string appears in the output membrane $i_f : q_kW((i_1, i_2), i_f)W$. 

Notice, that in region $m_i$ rule $1_j : ins_r(a_j, n_{ij})$ must be applied (otherwise string $SW$ appears in the skin by rule $2 : ins_r(S, \text{out})$ and will stay there forever). Moreover, this rule $1_j$ must be applied exactly once, after that rule $2$ applies and string $S\sigma a_k W$ is sent to the skin. At this point rule $2 : del_r(S, i_s)$ deletes $S$. Now this string is ready to continue the evolution. Notice that if rule $1_j$ were applied more than once, a string of the form $S\sigma a_k W'$ appears in the skin and the symbols $q_l$ at the right cannot be further deleted. By the given construction, the system $\Pi$ generates exactly the words accepted by $P$. Moreover, if a derivation of $\Pi$ differs from the pattern shown above, then it will contain nonterminals $q_i \notin \Sigma$ that cannot be removed and, hence, such word is not in $L(\Pi)$. Therefore, $L(P) = L(\Pi)$.

**Corollary 3.** Since the family of regular languages is closed with respect to taking the mirror image, the following relations hold:

$$ELSP(e - ins_{1,0}^0, e - del_{1,0}^0) \supseteq ELSP(r - ins_{1,0}^0, r - del_{1,0}^0) \supseteq \mathcal{REG}.$$ 

$$ELSP(e - ins_{1,0}^0, e - del_{1,0}^0) \supseteq ELSP(l - ins_{1,0}^0, l - del_{1,0}^0) \supseteq \mathcal{REG}.$$ 

### 4 Conclusions

In this paper we first introduced a tool — a simple computationally complete string rewriting model, NCPM5 (non-deterministic circular Post machines), for a purpose of allowing as simple simulation as possible by systems with insertion and deletion of prefixes and suffixes. Then we focus on distributed systems inserting and deleting substrings at the ends of the string, called P systems with exo-insertion and exo-deletion without contexts (eIDPs).

The focus of study is the maximal size of inserted and deleted substrings. It is known from [1] that eIDPs are computationally complete with either 1-symbol insertion and 2-symbol deletion, or with 2-symbol insertion and 1-symbol
deletion, or even with 1-symbol insertion and 1-symbol deletion in the tissue case. The general problem about 1-symbol insertion and 1-symbol deletion has been left open. In this paper, we have partially answered it, by showing that the computational completeness holds if we impose the priority of deletion rules over insertion rules.

We also gave a number of remarks related to the variants of CPMs and to the interpretations of computational completeness. With respect to the general problem above, in the final part of the paper we showed that the family of regular languages is the lower bound, by a construction with one-sided operations.

Acknowledgements A. Alhazov gratefully acknowledges the project RetroNet by the Lombardy Region of Italy under the ASTIL Program (regional decree 6119, 20100618). A. Krassovitskiy gratefully acknowledges the support of the Spanish Ministerio de Ciencia y Tecnología, project MTM2007-63422.

References


A Spiking Neural P system simulator based on CUDA

Francis Cabarle¹, Henry Adorna¹, Miguel A. Martínez–del–Amor²

¹ Algorithms & Complexity Lab
Department of Computer Science
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines
E-mail: fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph
² Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: mdelamor@us.es

Abstract. In this paper we present a Spiking Neural P system (SNP system) simulator based on graphics processing units (GPUs). In particular we implement the simulator using NVIDIA CUDA enabled GPUs. The massively parallel architecture of current GPUs is very suitable for the maximally parallel computations of SNP systems. We simulate a wider variety of SNP systems, after presenting a previous work on SNP system matrix representation which led to their simulation in GPUs, and the simulation algorithm included here. Finally, we compare and present the performance speedups of the CPU-GPU based simulator over the CPU only simulator.

Key words: Membrane Computing, Spiking Neural P systems, Parallel Computing, GPU Computing, CUDA

1 Introduction

Inspiration taken from nature can help us define new computing models, with the aim of providing efficient solutions to the limitations of conventional models of computation. In this respect, Membrane Computing, which is a research area initiated by Gheorghe Păun in 1998 [7], provides distributed, parallel, and non-deterministic computing models known as P systems. These models are basically abstractions of the compartmentalized structure and parallel processing of biochemical information in biological cells.

Many P system variants have been defined in literature, and many of them have been proven to be computationally complete. Moreover, several general classifications of P systems are considered depending on the level of abstraction: cell-like (a rooted tree where the skin or outermost cell membrane is the root, and its inner membranes are the children or leaf nodes in the tree), tissue-like (a
graph connecting the cell membranes) and neural-like (a directed graph, inspired by neurons interconnected by their axons and synapses). The last type refers to Spiking Neural P systems (in short, SNP systems), where the time difference (when neurons fire and/or spike) plays an essential role in the computations [14].

One key reason of interest for P systems is that they are able to solve computationally hard problems (e.g., NP-complete problems) usually in polynomial to linear time only, but requiring exponential space as trade off. These solutions are inspired by the capability of cells to produce an exponential number of new membranes via methods like mitosis (membrane division) or autopoiesis (membrane creation). However, because of this massively parallel, distributed and non-deterministic nature of P systems, they are yet to be fully implemented in vivo, in vitro, or even in silico. Thus, practical computations of P systems are driven by silicon-based simulators.

Since P systems were introduced, many simulators have been produced by using different software and hardware technologies [2]. In practice, P system simulations are limited by the physical laws of silicon architectures, which are often inefficient or not suitable when dealing with P system features, such as the exponential workspace creation and massive parallelism. However, in order to improve the efficiency of the simulators, it is necessary to exploit current technologies, leading to solutions in the area of High Performance Computing. In this way, many simulators have been developed over highly parallel platforms, including reconfigurable hardware as in FPGAs [16], CPU-based clusters [6], as well as the Graphical Processing Units (GPUs) [10,9]. These efforts show that parallel devices are very suitable in simulating P systems, at least for the first few P system variants to have been introduced (transition and active membrane P systems). Efficiently simulating SNP systems would thus require new attempts in parallel computing.

GPUs are the leading exemplars of modern high throughput-oriented architectures [12]. Nowadays, these kinds of processors are used for tackling problems where parallelism is prevalent. Indeed, GPUs have been successfully used to speedup many parallel applications. Modern GPUs are not limited only to graphics processing, as done by the first graphic cards, as they can now be used for general purpose computations [13]; they are now multi-core and data-parallel processors [3]. By using GPU computing, also known as GPGPU (General Purpose computation on the GPU), a programmer can achieve with a single GPU, a throughput similar to that of a CPU based cluster [13,15]. Thus, the main advantages of using GPUs are their low-cost, low-maintenance and low power consumption relative to conventional parallel clusters and setups, while providing comparable or improved computational power. Moreover, parallel computing concepts such as hardware abstraction, scaling, and so on are handled efficiently by current GPUs.

Given that SNP systems have already been represented as matrices due to their graph-like properties [17], simulating them in parallel devices such as GPUs is the next natural step. Matrix algorithms are well known in parallel computing
literature, including GPUs [11], due to the highly parallelizable nature of linear algebra computations mapping directly to the data-parallel GPU architecture.

Previously, SNP systems have been faithfully simulated in GPUs using their matrix representation [5]. This simulator combined both the object oriented programming language (OOPL) Python (CPU part) and CUDA/C (GPU part) codes, and so it has been improved [4], in performance, by using the PyCUDA [1] library. In this paper we present an extension of [4] in order to simulate SNP systems with more general regular expressions associated to their firing rules. This extension allows us to simulate larger and wider varieties of SNP systems in order to test the speedup achieved by the GPU and CPU based simulators.

This paper is organized as follows: Section 2 introduces SNP systems formally, as well as their matrix representation. Section 3 provides background for GPU computing with CUDA. The design of the simulator and simulation results are given in Section 4 and Section 5, respectively. Finally, conclusions, future work, acknowledgements, and references end this paper.

2 Spiking Neural P systems

In this section the SNP system model is introduced, together with a matrix representation of the model. This representation is the basis for the simulation algorithm in this paper. Additionally, the two examples used to test and analyse the simulator are also described.

2.1 The SNP systems model

Many variants of SNP systems have been introduced in recent works, such as those with delays, weights, extended firing rules, deterministic systems, division, budding, and so on. Each one has specific features in complexity, but the majority of them have been shown to be computationally complete [14, 8]. This paper is focused on a restricted variant of SNP systems, with no delays associated to the rules (i.e. neurons fire immediately once they are able to do so), which are of the following form:

Definition 1. An SNP system without delay, of degree $m \geq 1$, is a construct of the form

$$H = (O, \sigma_1, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the alphabet made up of only one object $a$, called spike;
2. $\sigma_1, \ldots, \sigma_m$ are $m$ neurons of the form

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$

where:

(a) $n_i \geq 0$ gives the initial number of spikes $a$ contained in neuron $\sigma_i$;
(b) $R_i$ is a finite set of rules of the following forms:

(b-1) $E/a^c \rightarrow a^p$, are known as Spiking rules, where $E$ is a regular expression over $a$, and $c \geq 1$, for $p \geq 1$ number of spikes are produced (with the restriction $c \geq p$), transmitted to each adjacent neuron with $\sigma_i$ as the originating neuron, and $a^c \in L(E)$;

(b-2) $a^s \rightarrow \lambda$, are known as Forgetting rules, for $s \geq 1$, such that for each rule $E/a^c \rightarrow a^p$ of type (b-1) from $R_i$, $a^s \notin L(E)$;

3. $\text{syn} = \{(i,j) \mid 1 \leq i,j \leq m, i \neq j\}$ are the synapses i.e. connections between neurons;

4. $\text{in}, \text{out} \in \{1,2,\ldots,m\}$ are the input and output neurons, respectively.

A spiking rule (type (b-1)) $r_s \in R_i$, where $1 \leq i \leq m$, is applied if the corresponding $\sigma_i$ contains $k$ spikes, $a^k \in L(E)$ and $k \geq c$. This means that consuming (removing) $c$ spikes (thus only $k-c$ spikes remain in $\sigma_i$), the neuron is fired, producing $p$ spikes that reach all $\sigma_j$ neurons immediately (no delays) such that $(i,j) \in \text{syn}$. An SNP system whose spiking rules have $p=1$ (they produce only one spike) is said to be of the standard type (non-extended). Forgetting rules (type (b-2)) are selected if $\sigma_i$ contains $s$ spikes. Thus, $s$ spikes are ‘forgotten’ or removed from the neuron once the rule is applied. Finally, a special case of (b-1) are rules of type (b-3) where $a^c \rightarrow a$, $L(E) = \{a^c\}$, $k = c, p = 1$. It is noteworthy that the neurons in an SNP system operate in parallel and in unison, under a global clock [14]. Similar to the usual way of using rules in other P system variants, there is maximal parallelism at the level of the system, in the sense that in each step all neurons which can spike (i.e. fire a rule) have to do it. However, only one rule can be applied at a given time in each neuron [14, 17]. The non-determinism of SNP systems comes with this fact: $L(E) \cap L(E') \neq \emptyset$ for two different spiking rules i.e. exactly one rule among several other applicable rules is chosen non-deterministically.

2.2 Examples of SNP systems

Next, two specific SNP systems are described to show their formal description based on Definition 1. The examples are from [17] and [14], and they are called $H_1$ and $H_2$ respectively. These examples are used in Section 5 to analyse the performance of the simulator.

The SNP system $H_1$ shown in Figure 1 generates all numbers in the set $\mathbb{N} - \{1\}$ (so it doesn’t halt) and the outputs of the computation are derived from the time difference between the first spike of the output neuron (to the environment) and its succeeding spikes. It can be seen that a total system ordering is given to neurons (from $\sigma_1$ to $\sigma_3$) and rules (from $R_1$ to $R_5$). This SNP system is of the form $H_1 = \{\{a\}, \sigma_1, \sigma_2, \sigma_3, \text{syn}, \text{out}\}$ where the first neuron $\sigma_1 = (2, \{R_1, R_2\})$, $n_1 = 2$, $R_1 = \{a^2/a \rightarrow a\}$, $R_2 = \{a^2 \rightarrow a\}$, $(\sigma_2$ to $\sigma_3$ and their $a, s$ and $R, s$ can be similarly shown), $\text{syn} = \{(1,2),(1,3),(2,1),(2,3)\}$ are the synapses for $H_1$, and the output neuron $\text{out} = \sigma_3$, as can be seen by the arrow not pointing to any neuron. $H_1$ has no input neuron.
Fig. 1. $\Pi_1$, an SNP system generating all numbers in the set $\mathbb{N} - \{1\}$, from [17].

A second SNP system $\Pi_2$, adapted from Figure 8 of [14], is shown in Figure 2. $\Pi_2$ is larger (in terms of the number of neurons and rules) than $\Pi_1$ (from Figure 1) and is formally defined as follows:

$$\Pi_2 = (\{a\}, \sigma_1, \sigma_{f_1}, \sigma_{f_2}, \sigma_{f_3}, \sigma_{f_4}, \sigma_{f_5}, \sigma_{f_6}, \sigma_{f_7}, \sigma_{f_8}, \sigma_{f_9}, \sigma_{out}, syn, out)$$

where the $\sigma$ (neuron) labelling are taken from [14].

Including the total ordering of the rules in $\Pi_2$, we have the following:

- $\sigma_1 = (0, \{R_1, R_2\})$, $n_1 = 0$ ($\sigma_1$ in this case initially has no spikes), $R_1 = a^3 (aa)^+ / a^2 \rightarrow a$, $R_2 = a^3 \rightarrow a$,
- $\sigma_{f_1} = (0, \{R_3\})$, $n_{f_1} = 0$, $R_3 = \{a \rightarrow a\}$,
- $\sigma_{f_2} = (0, \{R_4\})$, $n_{f_2} = 0$, $R_4 = \{a \rightarrow a\}$,
- $\sigma_{f_3} = (0, \{R_5, R_6\})$, $n_{f_3} = 0$, $R_5 = \{a^2 \rightarrow a\}$, $R_6 = \{a \rightarrow \lambda\}$,
- $\sigma_{f_4} = (0, \{R_7, R_8\})$, $n_{f_4} = 0$, $R_7 = \{a^2 \rightarrow a\}$, $R_8 = \{a \rightarrow \lambda\}$,
- $\sigma_{f_5} = (0, \{R_9\})$, $n_{f_5} = 0$, $R_9 = \{a \rightarrow a\}$,
- $\sigma_{f_6} = (0, \{R_{10}, R_{11}\})$, $n_{f_6} = 0$, $R_{10} = \{a^3 \rightarrow \lambda\}$, $R_{11} = \{a^2 \rightarrow 2\}$,
- $\sigma_{f_7} = (0, \{R_{12}, R_{13}\})$, $n_{f_7} = 0$, $R_{12} = \{a^2 \rightarrow a\}$, $R_{13} = \{a \rightarrow \lambda\}$,
- $\sigma_{out} = (0, \{R_{14}\})$, $n_{out} = 0$, $R_{14} = \{a \rightarrow a\}$

As with $\Pi_1$, it is evident that for $\Pi_2$ we have 17 synapses connecting the $\sigma$s.

$\Pi_2$ doesn’t have an input $\sigma$ but it has $out = \sigma_{out}$.

2.3 Matrix representation of SNP systems

In [17], a matrix representation of SNP systems without delays was introduced. By using this algebraic representation, it is easy to describe a simulation algorithm for computing configurations in SNP systems without delays by only using vector-matrix operations. The matrix representation inherently and conveniently involves another SNP system variant, the variant with extended firing rules where neurons can fire more than one spike at a given time [8]. Another convenient use for this algebraic representation is the possibility of performing backwards computation, from the current configuration to an earlier configura-
This representation makes use of the following vectors and matrix definitions:

*Configuration vector* $C_k$ is the vector containing all spikes in every neuron on the $k$th computation step/time, where $C_0$ is the initial vector containing all spikes in the system at the beginning of the computation. For $\Pi_1$ (the example in Figure 1) $C_0 = <2, 1, 1>$. For $\Pi_2$ we don’t have a default $C_0$ so we will assign several later, for the purpose of simulation. For this work, since we are not simulating SNP systems with delays, the steps focused on are purely computational and not time steps.

*Spiking vector* $S_k$ shows, at a given configuration $C_k$, if a rule is applicable (having value 1) or not (having value 0 instead). For $\Pi_1$ we have the spiking vector $S_k = <1, 0, 1, 1, 0>$ given $C_0$. Note that a second spiking vector, $S_k = <0, 1, 1, 1, 0>$, is possible and valid if we use rule (2) over rule (1) instead (but not both at the same time). *Validity* in this case means that only one among several applicable rules is used and thus represented in the spiking vector. We recall the applicability of rules from the definitions of (b-1) and (b-3). It is worth mentioning that there are several $S_k$s for every possible rule selection in the system. We cannot have an $S_k = <1, 1, 1, 0>$ because we cannot use rule (1) and rule (2) at the same time, hence this $S_k$ is invalid.

![Diagram](image-url)

*Fig. 2.* $\Pi_2$, a larger SNP system with 9 neurons and 14 rules, adapted from [14].
Spiking transition matrix $M_{SNP}$ is a matrix comprised of $a_{ij}$ elements where $a_{ij}$ is given as

Definition 2.

$$a_{ij} = \begin{cases} 
-c, & \text{rule } r_i \text{ is in } \sigma_j \text{ and is applied consuming } c \text{ spikes;} \\
p, & \text{rule } r_i \text{ is in } \sigma_s (s \neq j \text{ and } (s, j) \in \text{syn}) \\
& \text{and is applied producing } p \text{ spikes in total;} \\
0, & \text{rule } r_i \text{ is in } \sigma_s (s \neq j \text{ and } (s, j) \notin \text{syn}).
\end{cases}$$

In such a scheme, rows represent rules and columns represent neurons. As mentioned earlier, SNP systems with extended firing rules can be also represented by the spiking transition matrix.

A negative entry in the spiking transition matrix corresponds to the consumption of spikes. Thus, it is easy to observe that each row has exactly one negative entry, and each column has at least one negative entry [17]. For $\Pi_1$ and $\Pi_2$, the spiking transition matrices $M_{\Pi_1}$ and $M_{\Pi_2}$ are shown in the equations 1 and 2, respectively.

$$M_{\Pi_1} = \begin{pmatrix} 
-1 & 1 & 1 \\
-2 & 1 & 1 \\
1 & -1 & 1 \\
0 & 0 & -1 \\
0 & 0 & -2 
\end{pmatrix} \quad (1)$$

$$M_{\Pi_2} = \begin{pmatrix} 
-2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
-3 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & -1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & -2 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -2 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & -2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 
\end{pmatrix} \quad (2)$$

Finally, the following equation provides the configuration vector at the $(k+1)$th step, given the configuration vector $(C_k)$ and spiking vector $(S_k)$ at the $k$th step, and $M_{\Pi}$:

$$C_{k+1} = C_k + S_k \cdot M_{\Pi} \quad (3)$$
3 GPU Computing

High Performance Computing provides solutions for improving the performance of software applications by using accelerators or many-core processors. In this respect, Graphics Processing Units (GPUs) have been consolidated as accelerators thanks to their throughput-oriented highly-parallel architecture [12], as well as their low-power consumption and low-cost compared to other parallel clusters and setups. At the moment, for around $500, the latest GPUs of NVIDIA with 512 cores and with a performance comparable to a cluster of multi-core CPUs, is readily available at consumer electronics stores. The programming and architectural aspects of GPUs are described in this section.

3.1 Compute Unified Device Architecture (CUDA)

As many-core based platforms, GPUs are massively parallel processors which have high chip scalability in terms of processing units (core), and high bandwidth with internal GPU memories. The common CPU architectures are composed of transistors associated with different computing tasks: control, caching, DRAM, and ALU (arithmetic and logic). In contrast, only a fraction of the CPU’s transistors allocated for control and caching are used by GPUs, since far more transistors are devoted for ALU [3]. This architectural difference is a very distinct and significant reason why GPUs offer larger performance increase over CPU only implementation of parallel code working on large amounts of input data.

The programmability of GPUs has been focused on graphics, but using a type of parallel computing technique called GPGPU (General Purpose computation on GPUs), the large amount of internal cores can be used in parallel for accelerating the execution of data-parallel algorithms. In order to provide a straightforward, easy-to-learn, and scalable programming framework for GPUs, NVIDIA corporation (a well known manufacturer of graphics processors) introduced the Compute Unified Device Architecture (CUDA) in 2007 [3]. CUDA is a programming model and hardware architecture for general purpose computations in NVIDIA’s GPUs (G80 and newer family of GPUs)[3]. By extending popular languages such as C, CUDA allows programmers to easily create software that will be executed in parallel, avoiding low-level graphics and hardware primitives[15]. Among the other benefits of CUDA are abstracted and automated scaling: GPUs with more cores will make the parallelized code run faster than GPUs with fewer cores [15].

As seen in Figure 3, CUDA implements a heterogeneous computing architecture, where two different parts are often considered: the host (CPU side) and the device (GPU side). The host/CPU part of the code is responsible for controlling the program execution flow, allocating memory in the host or device/GPU, and obtaining results from the device by executing specific codes. The device (or devices if there are several GPUs in the setup) acts as a parallel co-processor to the host. The host outsources the parallel part of the program as well as the data to the device, since it is more suited to parallel computations than the host.
Fig. 3. Structure of the CUDA programming model, from [9].

The code to be executed in a GPU is written in CUDA C (CUDA extended ANSI C programming language). The parallel distribution of the execution units (threads) in CUDA can be split up into multiple threads within multiple thread blocks, each contained within a grid of (thread) blocks (see Figure 3). These grids belong to a single device/single GPU. Each device has multiple cores, each capable of running its own block of threads [3, 15]. A function known as a kernel function is one that is called from the host but executed in the device. Using kernel functions, the programmer can specify the GPU resources: the layout of the threads (from one to three dimensions) and the blocks (from one to two dimensions). GPUs with the same architecture as the one used in this work has a maximum number of threads per block equal to 512. The maximum size of each dimension of a thread block is \((512 \times 512 \times 64)\), pertaining to the \(x, y,\) and \(z\) dimensions of a block respectively. Lastly, the maximum size of each dimension of a grid of thread block is \((65535 \times 65535 \times 1)\) for the grid’s \(x, y,\) and \(z\) dimensions.

3.2 SNP system GPU simulation considerations

To successfully simulate SNP systems, the input files are file versions of \(C_k, S_k, M_{SNP}\), and a file \(r\) containing the list of rules \(R_i\). Since SNP systems involve
regular expressions (similar to other \(P\) system variants), string manipulation of the number of spikes in a neuron satisfying the regular expression \(E\) is involved. An OOPL such as Python is very well suited for string manipulation. For the computations involving linear algebra, since \(C_k, S_k, M_{SNP}\) only have integral values, the C programming language (which NVIDIA extended for their purposes as CUDA C) is well suited. Another reason for using C is because of this C language extension of CUDA. In actuality, only the kernel functions are written in C, and those functions are embedded within the Python code. A row-major ordering, a well known linear array input format where rows are placed one after another from top to bottom, was used to input the elements of \(M_{SNP}\). As an example, row-major ordering of \(M_{\Pi_1}\) is: 1, 1, 1, 2, 1, 1, 1, 1, 0, 0, 1, 0, 0, 2.

The current dichotomy of the CUDA programming model is that the data or inputs are loaded and ‘prepared’ in the host part, then they are moved to the device part and the parallel part of the process is executed on the data. Finally, the results of the parallel processing are moved back to the host for further processing. As seen from Figure 3 kernel functions are called from the host sequentially but once executed in the device, their blocks and the threads within these blocks are executed in parallel.

\(PyCUDA\) was chosen in order to fully utilize the speedup of CUDA as well as minimize development time, and is a Python programming interface to CUDA [1]. \(PyCUDA\) was developed by mathematician Andreas Klöckner for his dissertation for a more efficient parallel computing on CUDA using Python: safer in terms of memory handling, object cleanup (among others), and faster (in terms of development time via abstractions etc) [1].

4 Simulating SNP systems in CUDA

In this paper we designate as \(snpgpu-sim3\) our improved simulator, as compared to \(snpgpu-sim2\) from [4]. The simulation algorithm for \(snpgpu-sim3\), as with \(snpgpu-sim2\), is shown in Algorithm 1. Algorithm 1 shows which part of the algorithm (and hence the simulation) is run: whether in the host part or the device part, or both.

Firstly, the simulation inputs are loaded as mentioned in Subsection 3.2, as white space separated files (for \(C_k, S_k,\) and \(M\)) and using delimiters ‘\@’ and ‘&’ in \(r\) to delineate one rule from another in the same neuron, and from one neuron to another, respectively. For example, the file \(r\) pertaining to the rules of \(\Pi_1\) contains: aa 1 1@aa 2 1@aa 1 1@aa 1 1@aa 1 0. \(R_1\) of \(\Pi_1\) pertains to aa 1 1. In this encoding, the first part is the regular expression \(E\) (in this case, \(a^2\) or \(aa\)), the middle part is the number of spikes consumed (one spike) and lastly, the number of spikes produced (again, one spike).

In part II, the number of spikes in a neuron are checked if they satisfy the regular expression \(E\). In \(snpgpu-sim2\) only rules of the form (b-3), and not (b-1), were simulated, thus \(snpgpu-sim3\) can simulate more general SNP systems. A function created in Python, \(chkRegExp(\ regexp, \ spikNum\ )\) returns a boolean value of \(True\) if and only if the number of spikes given by \(spikNum\) (and hence
the number of spikes in a $\sigma$) are in the language generated by the regular expression $regexp$. Otherwise, function $chkRegExp$ returns a boolean $False$. Part II is responsible for generating all possible and valid $S_k$s out of the current $C_k$s and the rule file $r$. Note that not all possible $S_k$s (strings of 1s and 0s) are valid, since exactly one rule only, chosen non-deterministically, is applied per neuron per time step. The simulation ‘implements’ the non-determinism (as non-determinism is yet to be fully realized in hardware) by producing all the possible and valid $S_k$s for the given $C_k$s, and proceeds to compute each of the $C_{k+1}$ from these.

The process by which all possible and valid $S_k$s are produced is as follows: Once all the rules in the system are identified, given the current $n_i$s (number of spikes present in each of the $\sigma_i$s), the $\{1,0\}$ strings (at the moment they are treated as strings, and then as integral values later on) are produced on a per neuron level. As an example, given that $n_1 = 2$ for $\Pi_1$, and its two rules $R_1$ and $R_2$, we have the neuron-level strings ‘10’ (we choose to use $R_1$ instead of $R_2$) and ‘01’ (use $R_2$ instead of $R_1$). For $\sigma_2$ we only have ‘1’ ($R_3$ of $\sigma_2$ has the needed single spike, and it has only one rule) while $\sigma_3$ gives us ‘10’ since its single spike enables $R_4$ only and not $R_5$. After producing the neuron-level $\{1,0\}$ strings, the strings are exhaustively paired up, from left to right (since there is a need for ordering), until finally all the valid and possible $S_k$s from the current $C_k$s are produced. For $H_1$, given $C_0 = <2,1,1>$ we have $S_k$s $(1,0,1,1,0)$ and $(0,1,1,1,0)$.

Part III performs Equation 3, which is done in parallel in the device. The previously loaded values of $C_k$, $S_k$, $r$, and $M$ which were treated as strings (for the purposes of concatenation, regular expression checking, among others) are now treated as integral values. Each thread in the device contains either a matrix element from $M$ or a vector element from $S_k$ or $C_k$, and Equation 3 is performed in parallel. The newly produced $C_{k+1}$ are then moved back from the device to the host. Part IV then checks whether to proceed or to stop based on 2 stopping criteria for the simulation: (I) if a zero vector (vector of zeros) is encountered, (II) if the succeeding $C_k$s have all been produced in previous computations. Both (I) and (II) make sure that the simulation halts and does not enter an infinite loop.

**Algorithm 1** Overview of SNP system simulation algorithm

**Require**: Input files: $C_k$, $M$, $r$ (file counterparts of $C_k$, $M$, $R_i$).

I. **(HOST)** Load input files. $M$, $r$ are loaded once only. $C_0$ is also loaded once, then $C_k$s, $1 \leq k \leq m$, afterwards.

II. **(HOST)** Determine if a rule in $r$ is applicable based on the numbers of spikes present in each neuron/$\sigma$ seen in $C_k$. Then generate all valid and possible spiking vectors in a list of lists $S_k$ given the 3 inputs.

III. **(DEVICE)** Run kernel function on all valid and possible $S_k$s from the current $C_k$. Produce the next configurations, $C_{k+1}$ and their corresponding $S_k$s.

IV. **(HOST+DEVICE)** Repeat steps I to IV, till at least one of the two Stopping criteria is encountered.
In Figure 4 we see a graphical illustration of the simulation process, emphasizing the parts executed in the host/CPU and in the device/GPU. In the figure, it is assumed that there are $n$ number of $C_{k+1}$s produced from the current $C_k$. The CPU executes the simulation from top to bottom, calling the kernel function, and hence the GPU, in the third box. The smaller, multiple boxes in the device/GPU part illustrate the parallel computations of all the $C_{k+1}$s using Equation 3. Afterwards, the computed $C_{k+1}$s are sent back to the CPU which then decides what to do next, based on Algorithm 1, part IV.

Fig. 4. Diagram showing the simulation flow, with the host and device emphasized.

5 Simulation results and observations

Similar to the setup of $\text{snpgpu-sim}^2$, $\text{snpgpu-sim}^3$ simulated $H_1$ and $H_2$ using an Apple iMac running Mac OS X 10.5.8, with an Intel Core2Duo CPU at 2.66GHz and with a 6MB L2 cache. The GPU of the iMac is an NVIDIA GeForce 9400 graphics card at 1.15 GHz, with 256 MB Video RAM (or around $266 \times 10^6$ bytes), 16 cores, running CUDA version 3.1.

In order to compare the CPU-only SNP system simulator (we designate this as $\text{snpcpu-sim}$) the parallel parts of $\text{snpgpu-sim}^3$ were executed in a sequential manner. Hence, $\text{snpcpu-sim}$ runs entirely on the CPU only, while $\text{snpgpu-sim}^3$ uses both the CPU and the GPU of the iMac. Both simulators use Python and C, however only $\text{snpgpu-sim}^3$ use the CUDA enabled GPU. The simulations were done such that both $\text{snpcpu-sim}$ and $\text{snpgpu-sim}^3$ have the same $C_0$ or starting configuration as inputs. The simulations are run three times and the average
of the three trial runs is taken. The simulation comparison for $\Pi_1$ run in both simulators is shown in Figure 5.

![Fig. 5. Runtime graph of snpecpu-sim versus snpgpu-sim3 for $\Pi_1$.](image)

From Figure 5, the five $C_0$ values used in the simulation comparison are: (2,1,2), (3,1,3), (4,1,4), (6,1,6), and (9,1,9). The horizontal axis in Figure 5 are the $C_0$ values, while the vertical axis is the running time (in seconds) of the simulation on a given $C_0$. We see in Figure 5 that snpgpu-sim3 performs faster than snpecpu-sim given increasing values of $C_0$. A speedup of up to 1.4 times is achieved if $\Pi_1$ is run in snpgpu-sim3 instead of snpecpu-sim.

For $\Pi_2$, we introduce a variable $CkCnt$ in both simulators to limit the number of $C_k$s produced to a certain integer value so the simulation will not run indefinitely. Figure 6 shows the simulation of $\Pi_2$ for different values of $n$ (the number of spikes in $\sigma_1$ of $\Pi_2$ is $2n$) and $CkCnt$. In particular, the ($n$, $CkCnt$) pairs used are (5, 20), (10, 25), (15, 30), (20, 35), (50, 45), (70, 50), (100, 55), and (200, 60).

From Figure 6 we see that, for increasing values of $n$ and $CkCnt$, the simulation time of $\Pi_2$ in snpecpu-sim increases dramatically, unlike in snpgpu-sim3. The lack of a dramatic increase in simulation time in snpgpu-sim3 is because of the fact that $\Pi_2$ has more rules and neurons, and hence, exploits the parallel nature of the CUDA GPU all the more. A speedup of up to 6.8 times is achieved from the simulation on snpgpu-sim3 over snpecpu-sim.

We can calculate the maximum number of neurons snpgpu-sim3 can simulate using the 266x10^6 bytes of the NVIDIA GeForce 9400 GPU. Global memory used is $GbMem = 4*\text{sizeof}(Ck) + \text{sizeof}(M)$. Here we speak in terms of the sizeof operator in the standard C language since we use the data type int which is of size 4 bytes or 32 bits. The length of $Ck$ is the number of neurons, and $M$ is of size $R$ (rows are the number of rules) x $Ck$ (columns are number of neurons). We multiply $Ck$ to 4 because we also allocate memory for $Sk$, $Ck + 1$, etc.
MSk which temporarily holds the product of M and Sk, all of which are of the same size as Ck. We therefore have the following, substituting the real value of GbMem: \(266 \times 10^6\) (bytes) = 16(\text{bytes}) * Ck + 4(\text{bytes}) * R * Ck. Simplifying this equation and isolating Ck we have 
\[ Ck = \frac{266 \times 10^6\text{(bytes)}}{16\text{(bytes)} + 4(\text{bytes}) * R} \]

6 Conclusions and Future Work

In this paper we have simulated a wider variety of SNP systems using 
\textit{snpppu-sim3}, while maintaining the efficiency of the previous simulator 
\textit{snpppu-sim2}. The extension of the previous simulator \textit{snpppu-sim2}, presented here as \textit{snpppu-sim3}, can now simulate larger and wider varities of SNP systems by way of more general regular expressions (those of the form (b-1)). Two SNP systems were simulated: a basic one, \(\Pi_1\) (primarily for illustration purposes of the simulator) and a larger one (in terms of rules and neurons), \(\Pi_2\), to exemplify the speedup when using GPUs over CPU-only simulators. The speedup of \textit{snpppu-sim3} over \textit{snpcpu-sim} for \(\Pi_1\) went up to 1.4 times, while it was 6.8 for \(\Pi_2\). These results show that SNP system simulation on GPUs can greatly benefit from the parallel architecture of GPUs, and that increasing the parameters (in this case neurons and rules) offer even larger speedups. This benefit in speedup is coupled with the fact that the CUDA enabled graphics cards are readily available in consumer electronic stores. These cards offer boosts in general purpose computations as co-processors of commonly used CPUs, at a fraction of the power consumption of CPU clusters.

For future work, a generic P system parser based using P-lingua would be able to provide a more standardized input parser and formatting for the GPU based SNP system simulator. P-lingua is programming environment for Membrane computing which takes in as input a certain P system variant, and outputs
a standardized XML file format for the simulator to use. This standardized simulator input format would then be carried on to other simulators when simulating other P system variants.

Further understanding of the CUDA architecture as well as adapting the simulator to GPUs with more cores and their newer parallel technologies, as well as their subtleties, are also possibilities for future work, together with further improvements of the efficiency of the simulator. Lastly, more SNP system variants can be simulated by extending the current GPU simulator.

Acknowledgments

Francis Cabarle is supported by the DOST-ERDT program. Henry Adorna is funded by the DOST-ERDT research grant and the Alexan professorial chair of the UP Diliman Department of Computer Science. Miguel A. Martínez-del-Amor is supported by “Proyecto de Excelencia con Investigador de Reconocida Valía” of the “Junta de Andalucía” under grant P08-TIC04200, and by the project TIN2009–13192 of the “Ministerio de Educación y Ciencia” of Spain, both co-financed by FEDER funds.

References


Modularity in P Colonies with Checking Rules

Luděk Cienciala¹, Lucie Ciencialová¹, and Miroslav Langer¹

Institute of Computer Science, Silesian University in Opava, Czech Republic
{ludek.cienciala, lucie.ciencialova, miroslav.langer}@fpf.slu.cz

Abstract. We continue in the investigation of P colonies which were introduced in [9]. P colony is the abstract computing devices composed of independent agents, acting and evolving in a shared environment. The first part of the paper focuses on the modularity of the P colonies with checking rules. We deal with dividing the agents into modules. The second part of the paper focuses on the P colonies with capacity one. We introduce improved results of already presented results concerning the computational power of the P colonies with capacity one.

1 Introduction

P colonies were introduced in the paper [9] as formal models of a computing device inspired by membrane systems and formal grammars called colonies. This model is inspired by structure and function of a community of living organisms in a shared environment.

The independent organisms living in a P colony are called agents or cells. Each agent is represented by a collection of objects embedded in a membrane. The number of objects inside each agent is the same and constant during the computation.

The environment contains several copies of the basic environmental object denoted by $e$. The environmental object $e$ appears in arbitrary large number of copies in the environment.

A set of programs is associated with each agent. The program which determines the activity of the agent is very simple and depends on the content of the agent and on the multiset of objects placed in the environment. An agent can change the content of the environment by programs and it can affect the behaviour of other agents through the environment.

This interaction between agents is a key factor in functioning of the P colony. In each moment each object inside the agent is affected by executing the program.

For more information about P systems see [13] or [14].

2 Definitions

Throughout the paper we assume that the reader is familiar with the basics of the formal language theory.
We use $NRE$ to denote the family of the recursively enumerable sets of natural numbers. Let $\Sigma$ be the alphabet. Let $\Sigma^*$ be the set of all words over $\Sigma$ (including the empty word $\varepsilon$). We denote the length of the word $w \in \Sigma^*$ by $|w|$ and the number of occurrences of the symbol $a \in \Sigma$ in $w$ by $|w|_a$.

A multiset of objects $M$ is a pair $M = (V, f)$ where $V$ is an arbitrary (not necessarily finite) set of objects and $f$ is a mapping $f : V \to \mathbb{N}$; $f$ assigns to each object in $V$ its multiplicity in $M$. The set of all multisets with the set of objects $V$ is denoted by $V\circ$. The set $V'$ is called the support of $M$ and is denoted by $\text{supp}(M)$ if for all $x \in V'$ $f(x) \neq 0$ holds. The cardinality of $M$, denoted by $|M|$, is defined by $|M| = \sum_{a \in V} f(a)$. Each multiset of objects $M$ with the set of objects $V' = \{a_1, \ldots, a_n\}$ can be represented as a string $w$ over alphabet $V'$, where $|w|_{a_i} = f(a_i)$; $1 \leq i \leq n$. Obviously, all words obtained from $w$ by permuting the letters represent the same multiset $M$. $\varepsilon$ represents the empty multiset.

2.1 P colonies

We briefly recall the notion of P colonies. A P colony consists of agents and an environment. Both the agents and the environment contain objects. A set of programs is associated with each agent. There are two types of rules in the programs. The first type of rules, called the evolution rules, are of the form $a \rightarrow b$. It means that the object $a$ inside the agent is rewritten (evolved) to the object $b$. The second type of rules, called the communication rules, are of the form $c \leftrightarrow d$. If the communication rule is performed, the object $c$ inside the agent and the object $d$ outside the agent swap their places. Thus after executing the rule, the object $d$ appears inside the agent and the object $c$ is placed outside the agent.

In [8], the set of programs was extended by the checking rules. These rules give the agents an opportunity to opt between two possibilities. The rules are in the form $r_1/r_2$. If the checking rule is performed, then the rule $r_1$ has higher priority to be executed over the rule $r_2$. It means that the agent checks whether the rule $r_1$ is applicable. If the rule can be executed, then the agent is compulsory to use it. If the rule $r_1$ cannot be applied, then the agent uses the rule $r_2$.

Definition 1. The P colony of the capacity $k$ is a construct

$p = (A, e, f, V_E, B_1, \ldots, B_n)$, where

- $A$ is an alphabet of the colony, its elements are called objects,
- $e \in A$ is the basic object of the colony,
- $f \in A$ is the final object of the colony,
- $V_E$ is a multiset over $A - \{e\}$,
- $B_i$, $1 \leq i \leq n$, are agents, each agent is a construct $B_i = (O_i, P_i)$, where
  - $O_i$ is a multiset over $A$, it determines the initial state (content) of the agent, $|O_i| = k$,
  - $P_i = \{p_{i,1,}, \ldots, p_{i,k_i}\}$ is a finite multiset of programs, where each program contains exactly $k$ rules, which are in one of the following forms each:
An initial configuration of the P colony is an \((n + 1)\)-tuple of strings of objects present in the P colony at the beginning of the computation. It is given by the multiset \(O_i\) for \(1 \leq i \leq n\) and by the set \(V_E\). Formally, the configuration of the P colony \(\Pi\) is given by \((w_1, \ldots, w_n, V_E)\), where \(|w_i| = k\), \(1 \leq i \leq n\). \(w_i\) represents all the objects placed inside the \(i\)-th agent, and \(w_E \in (A - \{e\})^*\) represents all the objects in the environment different from the object \(e\).

In the paper parallel model of P colonies will be studied. That means that each agent tries to find one usable program at each step of the parallel computation. If the number of applicable programs is higher than one, then the agent chooses one of the programs nondeterministically. At one step of the computation the maximal possible number of agents is active. The program can be applied if all the conditions are met; i.e. there are appropriate objects within the environment and/or inside the agent.

A configuration is halting if the set of program labels \(P\) satisfying the conditions above cannot vary from the empty set. A set of all possible halting configurations is denoted by \(H\). A halting computation can be associated with the result of the computation. It is given by the number of copies of the special symbol \(f\) present in the environment. The set of numbers computed by a P colony \(\Pi\) is defined as

\[
N(\Pi) = \{ |v_E| \mid (w_1, \ldots, w_n, V_E) \Rightarrow^* (v_1, \ldots, v_n, v_E) \in H \},
\]

where \((w_1, \ldots, w_n, V_E)\) is the initial configuration, \((v_1, \ldots, v_n, v_E)\) is a halting configuration, and \(\Rightarrow^*\) denotes the reflexive and transitive closure of \(\Rightarrow\).

Consider a P colony \(\Pi = (A, e, f, V_E, B_1, \ldots, B_n)\). The maximal number of programs associated with the agents in the P colony \(\Pi\) is called the height of the P colony \(\Pi\). The degree of the P colony \(\Pi\) is the number of agents in it. The third parameter characterizing the P colony is its capacity describing the number of the objects inside each agent.

Let us use the following notations:

\(NPCOL_{par}(k, n, h)\) for the family of all sets of numbers computed by the P colonies working in a parallel, using no checking rules and with:
- the capacity at most \(k\),
- the degree at most \(n\) and
- the height at most \(h\).

If we allow the checking rules, then the family of all sets of numbers computed by the P colonies is denoted by \(NPCOL_{par}K\).

### 2.2 Register machines

The aim of the paper is to characterize the size of the families \(NPCOL_{par}(k, n, h)\) comparing them with the recursively enumerable sets of numbers. To meet the target, we use the notion of a register machine.
Definition 2. [10] A register machine is the construct \( M = (m, H, l_0, l_h, P) \) where:
- \( m \) is the number of registers,
- \( H \) is the set of instruction labels,
- \( l_0 \) is the start label, \( l_h \) is the final label,
- \( P \) is a finite set of instructions injectively labelled with the elements from the set \( H \).

The instructions of the register machine are of the following forms:

- \( l_1 : (ADD(r), l_2, l_3) \): Add 1 to the content of the register \( r \) and proceed to the instruction (labeled with) \( l_2 \) or \( l_3 \).
- \( l_1 : (SUB(r), l_2, l_3) \): If the register \( r \) stores the value different from zero, then subtract 1 from its content and go to instruction \( l_2 \), otherwise proceed to instruction \( l_3 \).
- \( l_h : HALT \): Stop the machine. The final label \( l_h \) is only assigned to this instruction.

Without loss of generality, it can be assumed that in each \( ADD \) instruction \( l_1 : (ADD(r), l_2, l_3) \) and in each conditional \( SUB \) instruction \( l_1 : (SUB(r), l_2, l_3) \), the labels \( l_1, l_2, l_3 \) are mutually distinct.

The register machine \( M \) computes a set \( N(M) \) of numbers in the following way: the computation starts with all registers empty (hence storing the number zero) and with the instruction labeled \( l_0 \). The computation proceeds by applying the instructions indicated by the labels, which is allowed by the content of registers. If it reaches the halt instruction, then the number stored at that time in the register 1 is said to be computed by \( M \) and hence it is introduced in \( N(M) \). (Because of the nondeterminism in choosing the continuation of the computation in the case of \( ADD \)-instructions, \( N(M) \) can be an infinite set.) It is known (see e.g. [10]) that in this way we can compute all sets of numbers which are Turing computable.

Moreover, we call a register machine partially blind [7] if we interpret a subtract instruction in the following way: \( l_1 : (SUB(r), l_2, l_3) \) - if there is a value different from zero in the register \( r \), then subtract one from its contents and go to instruction \( l_2 \) or to instruction \( l_3 \); if there is stored zero in the register \( r \) when attempting to decrement the register \( r \), then the program ends without yielding a result.

When the partially blind register machine reaches the final state, the result obtained in the first register is only taken into account if the remaining registers store value zero. The family of sets of non-negative integers generated by partially blind register machines is denoted by \( NRM_{pb} \). The partially blind register machine accepts a proper subset of \( NRE \).

3 Modularity in the terms of P colonies

During the evolution unicellular organisms have evolved into multicellular. Some cells specialized their activities for the particular function and have to cooperate with other specialized cells in order to be alive. In that way organs have evolved and living organisms have become more complex. Therefore the cooperating
organs and more complex living organisms are more sophisticated, live longer and their life is improving.

During the investigation of the P colonies we observed that some agents were providing the same function during the computation. This inspired us to introduce the modules in the P colonies. We have defined five modules where each of them is providing one specific function. These modules are the module for the duplication, the module for the addition, the module for the subtraction, the balance-wheel module, the control module (see Fig. 1). Definition of each module’s function is given in the proof of the following theorem. Introducing modules in the P colonies brings new approach to the computation algorithm, which allows us to improve already known results.

\[ \text{Theorem 1.} \quad \text{NPCOL}_{par}(1, 8, *) = \text{NRE}. \]

\[ \text{Theorem 2.} \quad \text{NPCOL}_{par}K(1, 7, *) = \text{NRE}. \]

\[ \text{Proof.} \quad \text{Let us consider a register machine } M \text{ with } m \text{ registers. We construct the P colony } \Pi = (A,e,f,V,E,B_1,B_2) \text{ simulating a computation of the register machine } M \text{ with:} \]

- \[ A = \{e,d,i,d',n,y\} \cup \{l_i,i,i',i'' | l_i \in H\} \cup \{a_r,A_r,S_r,B_r | 1 \leq r \leq m\}, \]
- \[ f = a_1, \]
- \[ B_i = (O_i,P_i), \quad O_i = \{e\}, i = 1,2 \]

We can group the agents of the P colony into five modules. Each module needs for its work an input and requires some objects. The result of its computation is an output:

1) the module for the duplication (uses 2 agents):

\[ P_1 : \quad P_2 : \quad P_3 : \]

\[
\begin{align*}
1 : & \langle e \leftrightarrow D_i \rangle, & 6 : & \langle d' \rightarrow i \rangle, & 10 : & \langle e \leftrightarrow i' \rangle, \\
2 : & \langle D_i \rightarrow i' \rangle, & 7 : & \langle i'' \leftrightarrow i' \rangle, & 11 : & \langle i' \rightarrow i'' \rangle, \\
3 : & \langle i' \leftrightarrow d \rangle, & 8 : & \langle i'' \rightarrow i \rangle, & 12 : & \langle i'' \leftrightarrow e \rangle, \\
4 : & \langle d \rightarrow d' \rangle, & 9 : & \langle i \leftrightarrow e \rangle, \\
5 : & \langle a \rightarrow d' \rangle,
\end{align*}
\]

Fig. 1. Modular P colony

\[ \text{Theorem 1.} \quad \text{NPCOL}_{par}(1, 8, *) = \text{NRE}. \]

\[ \text{Theorem 2.} \quad \text{NPCOL}_{par}K(1, 7, *) = \text{NRE}. \]
The duplicating module is activated when the object \( D_i \) appears in the environment. This object carries a message “Duplicate object \( i \)”. 

**Input:** one object \( D_i \)

**Output:** one object \( i \) after 9 steps and one object \( \square \) after 7 steps

**Requirements:** one object \( d \)

<table>
<thead>
<tr>
<th>Configuration of ( H )</th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>Env</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( e ) ( c ) ( dD_i )</td>
<td>1</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. ( D_i ) ( c ) ( d )</td>
<td>2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. ( i' ) ( e ) ( d )</td>
<td>3</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. ( d ) ( e ) ( i' )</td>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. ( \overline{d} ) ( i' )</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. ( d' ) ( i'' )</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. ( \square ) ( e ) ( i'' )</td>
<td>7</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. ( i'' ) ( e ) ( i )</td>
<td>8</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. ( i ) ( e ) ( i )</td>
<td>9</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. ( e ) ( e ) ( i )</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The duplicating module duplicates requested object.

(2) the module for the addition (uses 1 agent):

\[ P_1 : \]

1. \( \langle e \leftrightarrow A_r \rangle \),
2. \( \langle A_r \rightarrow a_r \rangle \),
3. \( \langle a_r \leftrightarrow e \rangle \).

**Input:** one object \( A_r \)

**Output:** one object \( a_r \) after 4 steps

**Requirements:** \( \emptyset \)

<table>
<thead>
<tr>
<th>Configuration of ( H )</th>
<th>( B_1 )</th>
<th>Env</th>
<th>( P_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( e ) ( A_r )</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. ( A_r )</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. ( a_r )</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. ( e ) ( a_r )</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The module for the addition adds one symbol into the environment.

(3) the module for the subtraction (uses 1 agent):

\[ P_1 : \quad P_1 : \]

1. \( \langle e \leftrightarrow S_r \rangle \),
4. \( \langle a_r \rightarrow y \rangle \),
2. \( \langle S_r \rightarrow B_r \rangle \),
5. \( \langle n \leftrightarrow e \rangle \).
3. \( \langle B_r \leftrightarrow a_r/B_r \rightarrow n \rangle \), 6. \( \langle y \leftrightarrow e \rangle \).

**Input:** one object \( S_r \)

**Output:** one object \( y \) after 5 steps or one object \( n \) after 4 steps

**Requirements:** object \( a_r \) (if there is at least one in the environment)
The part of computation when there is at least one $a_r$ in the environment configuration of $\Pi_B^1$ Env $P_1$.

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$E_{nu}$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $e$</td>
<td>$S_r a_r$</td>
<td>1</td>
</tr>
<tr>
<td>2. $S_r$</td>
<td>$a_r$</td>
<td>2</td>
</tr>
<tr>
<td>3. $B_r$</td>
<td>$a_r$</td>
<td>3</td>
</tr>
<tr>
<td>4. $a_r$</td>
<td>$B_r$</td>
<td>4</td>
</tr>
<tr>
<td>5. $y$</td>
<td>$B_r$</td>
<td>6</td>
</tr>
<tr>
<td>6. $e$</td>
<td>$B_r y$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

The part of computation when there is no copy of $a_r$ in the environment configuration of $\Pi_B^1$ Env $P_1$.

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$E_{nu}$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $e$</td>
<td>$S_r$</td>
<td>1</td>
</tr>
<tr>
<td>2. $S_r$</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3. $B_r$</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4. $n$</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5. $e$</td>
<td>$n$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

The module for the subtraction removes requested object from the environment.

(4) The balance-wheel module (uses 1 agent):

$P_1$:

1: $\langle e \rightarrow d \rangle$
2: $\langle d \leftrightarrow l_h / d \leftrightarrow e \rangle$

The balance-wheel module “keeps the computation alive”. It inserts the objects $d$ into the environment until it consumes a special symbol $f$ from the environment. This action makes it stop working. The object $f$ gets into the environment from the duplicating module which is activated by the simulation of the halt instruction by the control module.

(5) The control module (uses 2 agents):

a) initialization:

$P_1$:

1: $\langle e \rightarrow l_0 \rangle$

First agent in this module generates label of the first instruction of the register machine.

b) adding instruction $l_1$ : $(\text{ADD}(r), l_2, l_3)$:

$P_1$:

1: $\langle l_1 \rightarrow D_1 \rangle$, notes
2: $\langle D_1 \leftrightarrow d \rangle$, $\rightarrow$ duplication module
3: $\langle d \leftrightarrow \square \rangle$, $\leftarrow$ duplication module
4: $\langle \square \rightarrow A_r \rangle$, $\leftarrow$ duplication module
5: $\langle A_r \leftrightarrow 1 \rangle$, $\rightarrow$ addition module
6: $\langle 1 \rightarrow l_2 \rangle$, $\leftarrow$
7: $\langle 1 \rightarrow l_3 \rangle$, $\rightarrow$

The control module asks for duplication of the objects corresponding to the actual instruction. The first symbol is used for the activation of the Adding module and the second one is used to generate the object which corresponds to the label of the next instruction.
The control module asks for duplication of the objects corresponding to the actual instruction again. The first symbol is used for the activation of the subtracting module and the second one is used to check the generated object which was created according to the answer from the subtracting module. Finally it generates object which corresponds to the label of the instruction which register machine will execute in the corresponding configuration.
If the subtraction module generates $y$
configuration of $\Pi B_1 B_2 Env P_1 P_2$

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$Env$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.</td>
<td>$d$</td>
<td>$e$</td>
<td>$y1$</td>
<td>6</td>
</tr>
<tr>
<td>20.</td>
<td>$y$</td>
<td>$e$</td>
<td>$1d$</td>
<td>8</td>
</tr>
<tr>
<td>21.</td>
<td>$L'_1$</td>
<td>$e$</td>
<td>$1d$</td>
<td>10</td>
</tr>
<tr>
<td>22.</td>
<td>1</td>
<td>$e$</td>
<td>$L'_1d$</td>
<td>12</td>
</tr>
<tr>
<td>23.</td>
<td>$d$</td>
<td>$L'_1$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
<tr>
<td>24.</td>
<td>$d$</td>
<td>$l_2$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
<tr>
<td>25.</td>
<td>$d$</td>
<td>$e$</td>
<td>$dl_2$</td>
<td>13</td>
</tr>
<tr>
<td>26.</td>
<td>$l_2$</td>
<td>$e$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

If the subtraction module generates $n$
configuration of $\Pi B_1 B_2 Env P_1 P_2$

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$Env$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.</td>
<td>$d$</td>
<td>$e$</td>
<td>$n1$</td>
<td>7</td>
</tr>
<tr>
<td>19.</td>
<td>$n$</td>
<td>$e$</td>
<td>$d1$</td>
<td>9</td>
</tr>
<tr>
<td>20.</td>
<td>$L''_1$</td>
<td>$e$</td>
<td>$d1$</td>
<td>11</td>
</tr>
<tr>
<td>21.</td>
<td>1</td>
<td>$e$</td>
<td>$L''_1d$</td>
<td>12</td>
</tr>
<tr>
<td>22.</td>
<td>$d$</td>
<td>$L''_1$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
<tr>
<td>23.</td>
<td>$d$</td>
<td>$l_3$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
<tr>
<td>24.</td>
<td>$d$</td>
<td>$e$</td>
<td>$l_3d$</td>
<td>14</td>
</tr>
<tr>
<td>25.</td>
<td>$l_3$</td>
<td>$e$</td>
<td>$d$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

1) halting instruction $l_h$:

$P_1$:

$\langle l_h \leftrightarrow d \rangle \rightarrow$ balance-wheel module

The control module controls all the computation. In order to support other modules’ work, $t$ sends necessary objects into the environment.

The $P$ colony $\Pi$ correctly simulates any computation of the register machine $M$.

\hfill $\square$

4 \hspace{1em} $P$ colonies with one object inside the agent

In this Section we analyze the behaviour of $P$ colonies with only one object inside each agent of $P$ colonies. It means that each program is formed by only one rule, either the evolution rule or the communication rule or the checking rule. If all agents have their programs with the evolution rules, the agents “live only for themselves” and do not communicate with the environment.

Following results were proved:

- $NPCOL_{par}(1, 1, 2, *) \supset NRM_{pb}$ in [3].
- $NPCOL_{par}(1, 2, 3, *) = NRE$ in [3].

**Theorem 3.** $NPCOL_{par}(1, 3, *) = NRE$

**Proof.** We construct a $P$ colony simulating the computation of the register machine. Because there are only copies of $e$ in the environment and inside the agents, we have to initialize a computation by generating the initial label $l_0$. After generating the symbol $l_0$, the agent stops. It can continue its activity only by using a program with the communication rule. Two agents will cooperate in order to simulate the ADD and SUB instructions. Let us consider an $m$-register machine $M = \langle m, H, l_0, l_h, P \rangle$ and present the content of the register $i$ by the number of copies of a specific object $a_i$ in the environment. We construct the $P$ colony $\Pi = \langle A, e, f, \emptyset, B_1, \ldots, B_4 \rangle$ with:

- alphabet $A = \{l_i, L_i, L'_i, \bigcup D_i, S_i, S'_i, S''_i, y_i, n_i, \mid 0 \leq i \leq |H| \} \cup \{a_i, 1 \leq i \leq m \}$
- $f = a_1$. 

Modularity in $P$ Colonies with Checking Rules

101
(1) To initialize simulation of computation of $M$ we define the agent $B_1 = (e, P_1)$ with a set of programs:

$$P_1 : \begin{align*}
1 : \langle e \to \ell_0 \rangle; \\
2 : \langle e \to d \rangle; \\
3 : \langle d \leftrightarrow \ell_1 / d \leftrightarrow e \rangle;
\end{align*}$$

(2) We need an additional agent to generate a special object $d$. This agent will be working during whole computation. In each pair of steps the agent $B_2$ places a copy of $d$ to the environment. This agent stops working when it consumes the symbol which is generated by the simulation of the instruction $l_h$ from the environment.

$$P_2 : \begin{align*}
2 : \langle e \to d \rangle; \\
3 : \langle d \leftrightarrow \ell_1 / d \leftrightarrow e \rangle;
\end{align*}$$

The P colony $\Pi$ starts its computation in the initial configuration $(e, e, e, \varepsilon)$. In the first subsequence of steps of the P colony $\Pi$ only agents $B_1, B_2$ can apply their programs.

<table>
<thead>
<tr>
<th>step</th>
<th>configuration of $\Pi$</th>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$B_3$</th>
<th>$Env$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td>$e$</td>
<td>$e$</td>
<td>$e$</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>$l_0$</td>
<td>$d$</td>
<td>$e$</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>$l_0$</td>
<td>$e$</td>
<td>$e$</td>
<td>$d$</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

(3) To simulate the ADD-instruction $l_1 : (\text{ADD}(r), l_2, l_3)$, we use two agents $B_1$ and $B_3$ in the P colony $\Pi$. These agents help each other to add a copy of the object $a_r$ and the object $l_2$ or $l_3$ into the environment.

An instruction $l_1 : (\text{ADD}(r), l_2, l_3)$ is simulated by the following sequence of steps. Let the content of the agent $B_2$ be $d$. 

This pair of agents generates two objects. One object increases value of the particular register and the second one defines the instruction from which simulation will continue. One agent is not able to generate both objects corresponding to the simulation of one instruction, because at the moment of placing all of its content into the environment via the communication rules, it does not know which instruction it simulates. It nondeterministically chooses one of the possible instructions. Now it is necessary to check whether the agent has chosen the right instruction. For this purpose the second agent slightly changes first generated object. The first agent swaps this changed object for the new one generated only if it belongs to the same instruction. If this is not done successfully, the computation never stops because of absence of the halting object for the agent $B_2$. 

An instruction $l_1 : (\text{ADD}(r), l_2, l_3)$ is simulated by the following sequence of steps. Let the content of the agent $B_2$ be $d$. 

$\ldots$
(4) For each SUB-instruction \( l_1 : (\text{SUB}(r), l_2, l_3) \), the following programs are introduced in the sets \( P_1 \) and \( P_3 \):

\[
\begin{array}{cccc}
P_1 & P_1 & P_3 & P_3 \\
23 : (l_1 \rightarrow S_1) , & 26 : (S'_1 \rightarrow l_2) , & 28 : (e \leftrightarrow S_1) , & 31 : (S''_1 \leftrightarrow e) , \\
24 : (S_1 \leftrightarrow d) , & 27 : (S''_1 \rightarrow l_3) , & 29 : (S_1 \rightarrow S'_1) , & 32 : (a_r \rightarrow e) ; \\
25 : (d \leftrightarrow S'_1/d \leftrightarrow S''_1) , & & 30 : (S'_1 \leftrightarrow a_r/S'_1 \rightarrow S''_1) , & \\
\end{array}
\]

Agents \( B_1 \) and \( B_3 \) collectively check the state of particular register and generate label of following instruction.

The instruction \( l_1 : (\text{SUB}(r), l_2, l_3) \) is simulated by the following sequence of steps.

If the value in counter \( r \) is zero:

\[
\begin{array}{cccccc}
\text{step} & \text{configuration of } H & \text{applicable programs} & P_1 & P_2 & P_3 \\
\hline
1. & l_1 & d & e & d^v & & 23 \ or \ 12 \ or \ 13 \ or \ 3 \ or \ 26 \ or \ 27 \ or \ 28 \ or \ 29 \ or \ 30 \\
2. & S_1 & e & e & d^{v+1} & & 24 \ or \ 2 \ or \ 28 \ or \ 29 \ or \ 30 \\
3. & d & d & e & S_1d^v & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
4. & d & e & S_1 & d^{v+1} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
5. & d & d & S'_1 & d^{v+1} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
6. & d & e & S''_1 & d^{v+2} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
7. & d & d & S''_1 & d^{v+2} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
8. & S'_1 & e & e & d^{v+3} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
9. & l_1 & d & e & d^{v+3} & - \ or \ 29 \ or \ 30 \ or \ 31 \ or \ 2 \ or \ 31 \\
\end{array}
\]
If the register \( r \) stores a value different from zero:

<table>
<thead>
<tr>
<th>step</th>
<th>configuration of ( \Pi )</th>
<th>applicable programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( l_1 ) ( d ) ( e ) ( a_r^v d^v )</td>
<td>( P_1 ) ( 23 ) ( 3 ) ( - )</td>
</tr>
<tr>
<td>2</td>
<td>( S_1 ) ( e ) ( e ) ( a_r^v d^{v+1} )</td>
<td>( P_2 ) ( 24 ) ( 2 ) ( - )</td>
</tr>
<tr>
<td>3</td>
<td>( d ) ( d ) ( e ) ( S_1 a_r^v d^v )</td>
<td>( P_3 ) ( - ) ( 3 ) ( 28 )</td>
</tr>
<tr>
<td>4</td>
<td>( d ) ( e ) ( S_1 ) ( a_r^v d^{v+1} )</td>
<td>( - ) ( 2 ) ( 29 )</td>
</tr>
<tr>
<td>5</td>
<td>( d ) ( d ) ( S_1' ) ( a_r^v d^{v+1} )</td>
<td>( - ) ( 3 ) ( 30 )</td>
</tr>
<tr>
<td>6</td>
<td>( d ) ( e ) ( a_r ) ( S_1 a_r^{v-1} d^{v+2} )</td>
<td>( 25 ) ( 2 ) ( 33 )</td>
</tr>
<tr>
<td>7</td>
<td>( S_1' ) ( d ) ( e ) ( a_r^{u-1} d^{v+2} )</td>
<td>( 26 ) ( 3 ) ( - )</td>
</tr>
<tr>
<td>8</td>
<td>( l_2 ) ( e ) ( e ) ( d^{v+3} )</td>
<td>( - ) ( 2 ) ( - )</td>
</tr>
</tbody>
</table>

(5) The halting instruction \( l_h \) is simulated by the agent \( B_1 \) with a subset of programs:

\[
P_1 : (l_h \leftrightarrow d) .
\]

The agent places the object \( l_h \) into the environment, from where it can be consumed by the agent \( B_2 \) and by this the agent \( B_2 \) stops its activity.

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{step} & B_1 & B_2 & B_3 & \text{Env} & \text{configuration of } \Pi & P_1 & P_2 & P_3 \\
\hline
1 & l_h & e & e & d^v & d^v & 34 & 2 & - \\
2 & d & e & d & l_h d^v & - & 3 & - \\
3 & d & l_h & e & d^{v+1} & - & - & - \\
\end{array}
\]

The \( P \) colony \( \Pi \) correctly simulates computation of the register machine \( M \). The computation of \( \Pi \) starts with no object \( a_r \) placed in the environment in the same way as the computation of \( M \) starts with zeros in all registers. The computation of \( \Pi \) stops if the symbol \( l_h \) is placed inside the agent \( B_2 \) in the same way as \( M \) stops by executing the halting instruction labelled \( l_h \). Consequently, \( N(M) = N(\Pi) \) and because the number of agents equals four, the proof is complete.

In the following theorem we improve previous results, we decrease number of the programs in the agent. Also the following results were proved:

- \( NPCOL_{par} K(1, *, 7) = NRE \) in [2],
- \( NPCOL_{par} K(1, *, 8) = NRE \) in [1]

We improve here the first of these results.

**Theorem 4.** \( NPCOL_{par} K(1, *, 5) = NRE \)

**Proof.** We construct a \( P \) colony simulating the computation of the register machine. Because there are only copies of \( e \) in the environment and inside the agents, we have to initialize a computation by generating the initial label \( l_0 \). After generating the symbol \( l_0 \) this agent stops and it can start its activity only by using a program with the communication rule. Two agents will cooperate in order to simulate the ADD and SUB instructions.
Let us consider an m-register machine $M = (m,H,l_0,l_h,P)$ and present
the content of the register $i$ by the number of copies of a specific object $a_i$
in the environment. We construct the P colony $H = (A,e,f,\emptyset,B_1,\ldots,B_n)$, $n = |H| + 2$ where:

- alphabet $A = \{l_i,l_i',l_i'',i_i,\overline{i},D_i,l_i|0 \leq i \leq |H||\cup \{a_i|1 \leq i \leq m\} \cup \{e,d\}$,

- $f = a_1$,

- $B_i = (e,P_i)$, $1 \leq i \leq 4$.

(1) To initialize simulation of computation of $M$, we define the agent $B_1 = (e,P_1)$
with a set of programs:

$$P_1:$$
\begin{align*}
1 & : (e \rightarrow l_0), \\
2 & : (l_0 \leftrightarrow d);
\end{align*}

(2) We need an additional agent to generate a special object $d$. This agent will
be working during whole computation. In each pair of steps the agent $B_2$ places
a copy of $d$ to the environment.

$$P_2:$$
\begin{align*}
3 & : (e \rightarrow d), \\
4 & : (d \leftrightarrow l_0/d \leftrightarrow e);
\end{align*}

The P colony $H$ starts its computation in the initial configuration $(e,e,e,e,e)$. In
the first subsequence of steps of the P colony $H$, only the agents $B_1$ and $B_2$
can apply their programs.

<table>
<thead>
<tr>
<th>step</th>
<th>configuration of part of $H$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$e$</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>$l_0$</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>$l_0$ $e$ $d$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4.</td>
<td>$d$ $d$ $l_0$</td>
<td>–</td>
<td>4</td>
</tr>
</tbody>
</table>

(3) To simulate the ADD-instruction $l_1 : (ADD(r),l_2,l_3)$, there are two agents
$B_{l_1}$ and $B_{l_2}$ in the P colony $H$. These agents help each other to add one copy
of the object $a_r$ and the object $l_2$ or $l_3$ to the environment.

$$P_{l_1}:$$
\begin{align*}
5 & : (e \leftrightarrow l_1), \\
6 & : (l_1 \rightarrow L_1), \\
7 & : (L_1 \leftrightarrow e),
\end{align*}

$$P_{l_2}:$$
\begin{align*}
8 & : (e \rightarrow L'_1), \\
9 & : (L'_1 \leftrightarrow L_1), \\
10 & : (L_1 \rightarrow L''_1),
\end{align*}

$$P_{l_3}:$$
\begin{align*}
11 & : (L''_1 \leftrightarrow e), \\
12 & : (e \leftrightarrow L'_1), \\
13 & : (L'_1 \leftrightarrow a_r), \\
14 & : (a_r \leftrightarrow e), \\
15 & : (e \leftrightarrow L''_1), \\
16 & : (L''_1 \rightarrow l_2), \\
17 & : (L'_1 \rightarrow l_3), \\
18 & : (l_2 \leftrightarrow e), \\
19 & : (l_3 \leftrightarrow e);
\end{align*}

The instruction $l_1 : (ADD(r),l_2,l_3)$ is simulated by the following sequence
of steps. Let the content of the agent $B_2$ be $d$. 
(4) For each SUB-instruction $l_1 : (\textit{SUB}(r), l_2, l_3)$, the below mentioned programs are introduced in the sets $P_1^1, P_1^2$ and in the set $P_1^3$:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{step} & B_{l_1} & B_{l_2} & B_{l_3} & E_{\text{env}} & P_{l_1} & P_{l_2} \hline
1. & e & e & e & e & 5 & 8 & - & - \hline
2. & l_1 & L_1' & e & e & 6 & - & - & - \hline
3. & l_1 & L_1' & e & e & 7 & - & - & - \hline
4. & e & L_1' & e & e & L_1 & - & 9 & - \hline
5. & l_1 & e & e & L_1' & - & 10 & 12 & - \hline
6. & e & L_1'' & L_1 & e & - & 11 & 13 & - \hline
7. & e & e & a_r & e & L_1'' & - & - & 14 \hline
8. & e & e & e & L_1'' & e & - & - & 16 \hline
9. & e & e & e & l_2 & a_r & - & - & - \hline
10. & e & e & e & e & l_2a_r & - & - & - \hline
\end{array}
\]

The instruction $l_1 : (\textit{SUB}(r), l_2, l_3)$ is simulated by the following sequence of steps:

The following computation proceeds as follows (we do not consider the number of copies of the object $d$ in the environment):

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{step} & B_{l_1} & B_{l_2} & B_{l_3} & E_{\text{env}} & P_{l_1} & P_{l_2} \hline
1. & e & e & e & l_1a_r^u & 20 & - & - \hline
2. & l_1 & e & e & a_r^u & 21 & - & - \hline
3. & S_1 & e & e & a_r^u & 22 & - & - \hline
4. & e & e & e & S_1a_r^u & - & 23 & - \hline
5. & e & S_1 & e & a_r^u & - & 24 & - \hline
6. & e & a_r & e & S_1a_r^{u-1} & - & 25 & - \hline
7. & l_2 & e & S_1a_r^{u-1} & - & 26 & - & - \hline
8. & e & e & e & l_2S_1a_r^{u-1} & - & - & - \hline
\end{array}
\]

(5) The halting instruction $l_h$ is simulated by agent $B_2$ which consumes the object $l_h$ and that stops the computation.

The P colony $H$ correctly simulates the computation of the register machine $M$. The computation of the $H$ starts with no object $a_r$, which indicates the content of the register $r$, placed in the environment, in the same way as the computation in the register machine $M$ starts with zeros in all registers.
Then the agents simulate the computation by simulating ADD and SUB instructions. The computation of the P colony $\Pi$ stops if the symbol $l_h$ is placed inside the corresponding agent as well as the register machine $M$ stops by executing the halting instruction labelled $l_h$. Consequently, $N(M) = N(\Pi)$ and because the number of agents equals four, the proof is complete.

$\square$

5 Conclusion

In this paper we have proved that the P colonies with capacity $k = 1$ and degree $n = 3$ with checking/evolution programs are computationally complete. In Section 3 we have shown that the P colonies with capacity $k = 1$ and degree $n = 7$ with checking/evolution programs are computationally complete. Although the this result is worse than the previous result, the main contribution of this section is grouping the agents into the modules. We have also shown, that the P colonies with capacity $k = 1$ and height $h = 5$ with checking/evolution programs are computationally complete. The generative power of $NPCOL_{par}K(1,n,2)$ remains open.

Remark 1. This work has been supported by the Grant Agency of the Czech Republic grants No. 201/06/0567, SGS/5/2010 and SGS/7/2011.

References

14. P systems web page: http://psystems.disco.unimib.it
A Connection Between Finite dP Automata and Multi-head Finite Automata

Erzsébet Csuhaj-Varjú** and György Vaszil

Computer and Automation Research Institute, Hungarian Academy of Sciences
Kende u. 13-17, H-1111 Budapest, Hungary
{csuhaj,vaszil}@sztaki.hu

Abstract. We describe a connection between dP automata (distributed P automata) and non-deterministic multi-head finite automata. We introduce the concepts of weak agreement language and strong agreement language of a dP automaton and the notion of a two-way dP automaton. Then we demonstrate how the language of a non-deterministic one-way multi-head finite automaton and the language of a non-deterministic two-way multi-head finite automaton can be obtained as the weak agreement language or the strong agreement language of a one-way, i.e., the usual finite dP automaton and a two-way finite dP automaton.

1 Introduction

Multi-head finite automata were introduced in the middle of the sixties of the last century [13], [14], so these natural extensions of the concept of a finite automaton have been studied for a long time. Unlike standard finite automaton, a multi-head finite automaton may have more than one heads reading the same input word; the heads may scan the input symbol and move when the state of the automaton changes. Acceptance is defined as in the one-head case: an input string is accepted if starting from the beginning of the word with all heads (that never leave the input word), the automaton enters an accepting state. We note that analogously to the one-head case, deterministic and non-deterministic, one-way and two-way variants are considered. (If the heads are allowed to move in both directions, the automaton is called two-way, if only from left to right, then one-way.)

Several restrictions and generalizations of multi-head finite automata have been investigated [16], leading to descriptions of interesting and important language and complexity classes. With one-head finite automata, irrespectively of being one-way or two-way, deterministic or non-deterministic, only the class of regular languages can be obtained. Increasing the number of the heads, the situation changes: the computational power of one-way finite automata with $k + 1$
heads is larger than that of one-way finite automata with $k$ heads for both
the deterministic and the non-deterministic variants. A similar result holds for
multi-head two-way automata: for each $k \geq 1$ there exists a language which can
be accepted by some deterministic (non-deterministic) finite automaton with
$k + 1$ heads but cannot be accepted by any deterministic (non-deterministic)
automaton with $k$ heads. Another important result is that the class of languages
accepted by two-way deterministic and two-way non-deterministic multi-head
automata are the complexity classes of deterministic and non-deterministic log-
arithmic space. For a survey on the history and results in the area, the reader is
referred to [7].

One interesting direction of research is to find counterparts of multi-head fi-
nite automata in other areas of computer science. In this paper we demonstrate
a connection between constructs called dP automata (distributed P automata)
and multi-head finite automata. We introduce the concepts of the weak agree-
ment language and the strong agreement language of a dP automaton and the
notion of a two-way dP automaton. Then we demonstrate how the language of
a non-deterministic one-way multi-head finite automaton and the language of a
non-deterministic two-way multi-head finite automaton can be obtained as the
weak agreement language or the strong agreement language of a one-way, i.e.,
the customary finite dP automaton and a two-way finite dP automaton.

The importance of these results lies in building a further bridge between a
classic, highly elaborated field of automata theory and the theory of P automata
which is a research field in membrane computing dealing with automata-like
unconventional computing devices.

P automata are variants of antiport P systems accepting strings in an autom-
aton-like fashion. They were introduced in [3] (for a summary on P automata,
see Chapter 6, [12]). Strings in the language of a P automaton are obtained as
mappings of the multiset sequences which enter the system through the skin
membrane during an accepting computation. The case of simple, non-erasing
mappings was studied in [2]. It was shown that if the rules of the P automaton are
applied sequentially, then the accepted language class is strictly included in the
class of languages accepted by one-way Turing machines with a logarithmically
bounded workspace, or, if the rules are applied in the maximally parallel manner,
then the class of context-sensitive languages is obtained.

The notion of a distributed P automaton (dP automaton in short) was intro-
duced in [9]. Such a system consists of a finite number of component P automata
which have their separate inputs and which also may communicate with each
other by means of special antiport-like rules. A string accepted by a dP automa-
ton is defined in [9] as the concatenation of the strings accepted by the individual
components during a computation performed by the system.

Since a multiset of objects can be represented as the set of all permutations
of its elements (a set of strings of objects), there are variants of P automata
which define their accepted languages by the mapping which maps any multiset
to the set of permutations of its objects. The generic variant in [9] also uses this
mapping, that is, a string accepted by a component P automaton is the concate-
nation of strings which are permutations of the objects of the multisets imported by the skin membrane during an accepting computation; all combinations are considered.

The computational power of dP automata was studied in [9], [5], [10], and [11]. Among other things, it was shown that dP automata with the above described permutation mapping are strictly more powerful than P automata, but the language family accepted by them is strictly included in the family of context-sensitive languages.

This paper is based on the observation that the way of functioning of these constructs is similar to that of multi-head automata: the components can represent heads, the multisets entering the components from the environment correspond to the strings scanned by the heads, and the states of the multi-head finite automata can be represented by configurations of the dP automaton. Since the heads of the multi-head finite automaton read the same input word, we should require that the components of the dP automaton recognize the same strings. For this reason, we introduce the notion of the agreement language, i.e., the set of strings which can be accepted by every component simultaneously during some accepting computation of the system. We note that we distinguish two types of agreement languages, the weak and the strong variants.

Two-way movement can also be interpreted in terms of dP automata, similarly to the approach of [1], if we consider a so-called double alphabet of objects consisting of barred and non-barred letters with a bijection between the two subalphabets. When a non-barred letter enters a component from the environment, it represents a move to the right of the corresponding head after reading the symbol, if a barred letter enters the system, then a move to the left of the corresponding head is assumed. Any two-way movement of the heads of a multi-head finite automaton can be described by strings over double alphabets, so based on these considerations, we may introduce the concept of a two-way dP automaton.

In the following sections we describe the connections between languages of non-deterministic one-way and two-way multi-head finite automata and agreement (weak and strong) languages of finite dP automata. Thus, we build a further bridge between classical and unconventional automata theory. We also discuss the concepts and the results, and propose further problems and research directions for future study.

2 Preliminaries

Throughout the paper we assume that the reader is familiar with the basics of formal language theory and membrane computing; for details we refer to [15] and [12].

An alphabet is a finite non-empty set of symbols. Given an alphabet $V$, we denote by $V^*$ the set of all strings over $V$. If the empty string, $\lambda$, is not included, then we use notation $V^+$. The length of a string $x \in V^*$ is denoted by $|x|$. For any set of symbols $A \subseteq V$ and for any symbol $a \in V$, the number of occurrences of symbols from $A$ in $x$ is denoted by $|x|_A$, while $|x|_a$ denotes the number of
occurrences of the symbol \( a \) in \( x \). The reverse (or the mirror image) \( x^R \) of a string \( x = x_1x_2 \ldots x_n, x_i \in V, 1 \leq i \leq n \), is defined as \( x^R = x_nx_{n-1} \ldots x_1 \), and \( \lambda^R = \lambda \).

A finite multiset over an alphabet \( V \) is a mapping \( M : V \rightarrow N \) where \( N \) is the notation for the set of non-negative integers; \( M(a) \) is said to be the multiplicity of \( a \) in \( M \). \( M \) can also be represented by all permutations of a string \( x = a_1^{M(a_1)}a_2^{M(a_2)} \ldots a_n^{M(a_n)} \in V^* \), where \( a_j \in V, M(a_j) \neq 0, 1 \leq j \leq n \). The set of all finite multisets over an alphabet \( V \) is denoted by \( V^+ \), and we use the notation \( V^{(+)} \) to denote the set of nonempty (finite) multisets. If no confusion arises, the empty multiset is denoted by \( \lambda \) as in the case of the empty string; otherwise we use \( \lambda \). We note that the above notations slightly differ from the customary notations in \( P \) systems theory, we use them to avoid confusion when both \( V^+ \) and \( V^{(+)} \) appear in the same context, that is, when we explicitly need to distinguish between strings and multisets.

We first recall some basic notions concerning multi-head finite automata, our presentation is based on [7]. A non-deterministic two-way \( k \)-head finite automaton (a 2NFA\((k)\), for short) is a construct \( M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, F) \), where \( Q \) is the finite set of states, \( \Sigma \) is the set of input symbols, \( k \geq 1 \) is the number of heads, \( \triangleright \not\in \Sigma \) and \( \triangleleft \not\in \Sigma \) are the left and the right endmarkers, respectively, \( q_0 \) is the initial state, \( F \subseteq Q \) is the set of accepting states, and \( \delta \) is the partial transition function which maps \( Q \times (\Sigma \cup \{\triangleright, \triangleleft\})^k \) into subsets of \( Q \times \{-1,0,1\}^k \), where \( 1 \) means that the head moves one tape cell (position) to the right, \( -1 \) means that it moves one tape cell to the left and \( 0 \) means that it remains at the same position. Whenever \((q', (d_1, \ldots, d_k)) \in \delta(q, (a_1, \ldots, a_k))\) is defined, then \( d_i \in \{0,1\} \) if \( a_i = \triangleright \) and \( d_i \in \{-1,0\} \) if \( a_i = \triangleleft \), \( 1 \leq i \leq k \), i.e., the heads can never move to the left from the left endmarker and to the right from the right endmarker.

A configuration of a 2NFA\((k)\) \( M = (Q, \Sigma, k, \triangleright, \triangleleft, q_0, F) \) is a triplet \( c = (w, q, p) \), where \( w \in \Sigma^* \) is the input, \( q \in Q \) is the current state, and \( p = (p_1, \ldots, p_k) \in \{0, 1, \ldots, |w| + 1\}^k \) gives the current head positions. If a position \( p_i \) is 0, then head \( i \) is scanning symbol \( \triangleright \), if \( 1 \leq p_i \leq |w| \), then head \( i \) scans the \( p_i \)th letter of \( w \), and if it is \( |w| + 1 \), then the head is scanning symbol \( \triangleleft \).

The initial configuration for an input \( w \) is \((w, q_0, (1, \ldots, 1))\), that is, a 2NFA\((k)\) starts computing the word with all of its heads on the first tape cell of the tape.

In the course of the computation, \( M \) performs direct changes of its configurations. Let \( w = a_1 \ldots a_n \), be the input, \( a_0 = \triangleright \), \( a_{n+1} = \triangleleft \). For two configurations \( c_1 = (w, q, (p_1, \ldots, p_k)) \) and \( c_2 = (w, q', (p'_1, \ldots, p'_k)) \) we say that \( c_2 \) directly follows \( c_1 \), denoted by \( c_1 \vdash c_2 \), if \( p'_i = p_i + d_i \), \( 1 \leq i \leq k \), where \((q', (d_1, \ldots, d_k)) \in \delta(q, (a_1, \ldots, a_k))\). The reflexive transitive closure of \( \vdash \) is denoted by \( \vdash^+ \). Note that due to the restriction of the transition function, the heads cannot move beyond the endmarkers.

The language \( L(M) \) accepted by a 2NFA\((k)\) \( M \) is the set of words \( w \) such that there is a computation which starts with \( \triangleright w \triangleleft \) on the input tape and ends when \( M \) reaches an accepting state, i.e.,

\[
L(M) = \{ w \in \Sigma^* \mid (w, q_0, (1, \ldots, 1)) \vdash^+ (w, q_f, (p_1, \ldots, p_k)), q_f \in F \}.
\]
If $\delta$ is a partial transition function mapping $Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k$ into subsets of $Q \times \{0, 1\}^k$, i.e., if the heads never move to the left, then $M$ is said to be non-deterministic one-way $k$-head finite automaton, a 1NFA($k$), for short. The class of languages accepted by 1NFA($k$) and 2NFA($k$), for $k \geq 1$, is denoted by $\mathcal{L}(1\text{NFA}(k))$ and $\mathcal{L}(2\text{NFA}(k))$, respectively.

Throughout the paper, we consider non-deterministic two-way (one-way) multi-head finite automata, therefore, it is clear from the context, we will not indicate the term non-deterministic.

**Remark 1.** Due to the definition of a 2NFA($k$), after reading the input symbol, the head do not need to move away from the tape cell containing the symbol. For technical reasons, we will modify the definition in such way that the automaton reads the input symbols only in the case when the head moves away from the tape cell containing this symbol. We define the transition function of a 2NFA($k$) as $\delta : Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ where for any $(q, (d_1, \ldots, d_k)) \in \delta(s, a_1, \ldots, a_k)$ if and only if $d_j = 0$ for some $j$, $1 \leq j \leq k$, then $a_j = \lambda$.

To see that the two definitions are equivalent, first note that for any $\delta : Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ we can construct a mapping $\delta' : Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ by defining $\delta'(s, a_1, \ldots, a_k) = \delta(s, a_1, \ldots, a_k)$ for all $s \in Q$ and $a_i \neq \lambda$, $1 \leq i \leq k$, and by constructing for each transition $(q, (d_1, \ldots, d_k)) \in \delta(s, a_1, \ldots, a_k)$ with $a_j = \lambda$, $d_j = 0$ for some $j$, $1 \leq j \leq k$, a set of transitions $(q, (d_1, \ldots, d_k)) \in \delta'(s, a_1', \ldots, a_k')$ for all input symbols $a_j' \in \Sigma$ and $a_i' = a_i$ if $i \neq j$, $1 \leq i \leq k$.

Furthermore, for $\delta : Q \times (\Sigma \cup \{\triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ we can construct an equivalent automaton with $\delta' : Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ as follows. Let $Q' = \{q, a_1, \ldots, a_k\}$ for all $(\alpha_1, \ldots, \alpha_k) \in (\Sigma \cup \{\triangleright, \triangleleft\})^k$. Let the initial state of the new automaton be $(q_0, * \ldots, *)$ and for any $(q, (d_1, \ldots, d_k)) \in \delta(s, a_1, \ldots, a_k)$ we define the transition $(q, (\beta_1, \ldots, \beta_k), (d_1, \ldots, d_k)) \in \delta'(q, \alpha_1, \ldots, \alpha_k, a_1', \ldots, a_k')$ with the following properties. If for some $j$, $1 \leq j \leq k$, (1) $\alpha_j = \ast$ and $d_j \neq 0$, then $\beta_j = \ast$ and $a_j' = a_j$; (2) $\alpha_j = \ast$ and $d_j = 0$, then $\beta_j = a$, $a \in \Sigma$, and $a_j' = \lambda$; (3) $\alpha_j \neq \ast$ and $d_j \neq 0$, then $\beta_j = \ast$ and $a_j' = a_j$; (4) $\alpha_j \neq \ast$ and $d_j = 0$, then $\beta_j = \alpha_j$ and $a_j' = \beta_j$.

It can easily be verified that the equivalence holds.

**Remark 2.** We will simplify the notation for the elements of the transition relation $\delta : Q \times (\Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^{Q \times \{0, 1\}^k}$ defined as above. Instead of $(q, (d_1, \ldots, d_k)) \in \delta(s, a_1, \ldots, a_k)$ with $d_i \in \{-1, 0, 1\}$ for $1 \leq i \leq k$, where $d_j = 0$ if and only if $a_j = \lambda$, we will have $\delta' : Q \times (\Sigma \cup \Sigma \cup \{\lambda, \triangleright, \triangleleft\})^k \rightarrow 2^Q$ where $\Sigma = \{a \mid a \in \Sigma\}$, and for any $(q, (d_1, \ldots, d_k)) \in \delta(s, a_1, \ldots, a_k)$ we will write $q \in \delta'(s, a_1', \ldots, a_k')$ where if $d_i = 1$, then $a_i' = a_i$; if $d_i = -1$ then $a_i' = \bar{a}_i$; and if $d_i = 0$, then $a_i' = a_i = \lambda$, $1 \leq i \leq k$.

In the following we briefly recall some notions from the theory of P systems we will use in the sequel; for further information consult the monograph [8] or the handbook [12].
A P system is a structure of hierarchically embedded membranes, each having a unique label and enclosing a region containing a multiset of objects. An antiport rule is of the form \((u, in; v, out)\), \(u, v \in O^{(n)}\) for a finite set of objects \(O\). If such a rule is applied in a region, then the objects of \(u\) enter from the parent region and in the same step, objects of \(v\) leave to the parent region. Antiport rules can also be equipped with promoters, written as \((u, in; v, out)_z, z \in O^{(n)}\). In this case the rule can only be applied if the objects of the promoter multiset \(z\) are all present in the child region.

A P automaton (of degree \(k\)) (introduced in [3], for details see [4]) is a membrane system \(\Pi = (O, \mu, P_1, \ldots, P_k, c_0, F)\) with object alphabet \(O\), membrane structure \(\mu\), sets of antiport rules (possibly with promoters) \(P_i, 1 \leq i \leq k\), initial configuration \(c_0 = (\mu, w_1, \ldots, w_k)\) where \(w_i \in O^{(i)}\), \(1 \leq i \leq k\), is the initial contents of the \(i\)th region, and set of accepting configurations \(F\) of the form \((\mu, v_1, \ldots, v_k), v_i \in O^{(i)}, 1 \leq i \leq k\).

The configurations of the P automaton are changed by applying the rules in the non-deterministic maximally parallel manner. This means that in every region a multiset of rules is applied, these rules are applied simultaneously in all the regions, and there is no more applicable rule which can be added to any of these multisets. This way, there is a sequence of multisets which enter the system from the environment during the steps of its computations. If the computation is accepting, that is, if it ends in an accepting configuration from \(F\), then this multiset sequence is called an accepted multiset sequence. The language accepted by the P automaton with respect to a mapping \(f : O^{(n)} \rightarrow 2^{O^{(n)}}\) is the \(f\)-image of the set of multiset sequences accepted by the P automaton.

If the P automaton accepts by final states, then \(F\) is given as \(E_1 \times \cdots \times E_k, E_i \subseteq O^{(i)}\), such that \(E_i\) is either a finite set of finite multisets, or \(E_i = O^{(i)}\), \(1 \leq i \leq k\). If \(\Pi\) accepts by halting, then \(F\) consists of all halting configurations of \(\Pi\), that is, of all configurations in which no rule can be applied in any of the regions.

A collection of P automata forms a distributed P automaton, a dP automaton, for short. Here we present the notion of a dP automaton in a slightly modified form as it was introduced in [9], in order to make it conform with the notations used for P automata in [4].

A dP automaton (of degree \(n \geq 1\)) is a construct \(\Delta = (\Pi_1, \ldots, \Pi_n, R, F)\), where \(O\) is an alphabet, the alphabet of objects; \(\Pi_i = (O, \mu_i, P_{i,1}, \ldots, P_{i,k_i}, c_{i,0}, F_i)\) is a P automaton of degree \(k_i \geq 1, 1 \leq i \leq n\), called the \(i\)th component of the system; \(R\) is a finite set of rules of the form \((s_i, u/v, s_j)\), \(1 \leq i, j \leq n, i \neq j, uv \in O^{(n)}\), called the set of inter-component communication (shortly, communication) rules of \(\Delta\); \(s_{k_i}, 1 \leq k \leq n\) denotes the skin membrane of \(\Pi_k\); \(F \subseteq F_1 \times \cdots \times F_n\), is called the set of accepting configurations of \(\Delta\).

An inter-component communication rule \((s_i, u/v, s_j), 1 \leq i, j \leq n, i \neq j\), realizes a direct communication between components \(\Pi_i\) and \(\Pi_j\): a multiset \(u\) in the skin region of \(\Pi_i\) is exchanged with a multiset \(v\) in the skin region of \(\Pi_j\).

A configuration of \(\Delta\) is \((\mu_1, u_{1,1}, \ldots, u_{1,k_1}), \ldots, (\mu_n, u_{n,1}, \ldots, u_{n,k_n})\), where \(u_{i,j}, 1 \leq i \leq n, 1 \leq j \leq k_i\), is a multiset over \(O\). The initial configuration of \(\Delta\)
is the n-tuple \(((\mu_1, w_{1,1}, \ldots, w_{1,k_1}), \ldots, (\mu_n, w_{n,1}, \ldots, w_{n,k_n})) = (c_{1,0}, \ldots, c_{n,0})\)
where \(c_{i,0}, 1 \leq i \leq n\), is the initial configuration of component \(\Pi_i\).

The dP automaton functions by changing its configurations. The components work synchronously, governed by a global clock, using the rules from their own rule sets and the set of communication rules \(R\) in the non-deterministic maximally parallel manner. Each component \(\Pi_i\), \(1 \leq i \leq n\), takes an input (may be the empty multiset) from the environment, works on it by using the rules in sets \(P_{i,1}, \ldots, P_{i,k_i}\) and possibly communicates with the other components. The communication is done by means of rules in \(R\).

A configuration \(C\) changes to configuration \(C'\) by importing the n-tuple of multisets \((u_1, \ldots, u_n)\) from the environment, denoted by \((u_1, \ldots, u_n, C) \Rightarrow C'\), if \(C'\) can be obtained from \(C\) by applying the rule sets of \(\Delta\) (including \(R\)) such that the skin membrane of component \(\Pi_i\) takes multiset \(u_i\) from the environment, i.e., \(u_i\) enters \(\Pi_i\) from the environment, \(1 \leq i \leq n\).

A computation in \(\Delta\) is a sequence of configurations following each other, starting from the initial configuration. It is successful if it enters one of the accepting configurations of \(F \subseteq F_1 \times \cdots \times F_n\). If the components accept by final states, then \(F = F_1 \times \cdots \times F_n\), or if \(\Delta\) accepts by halting, then \(F \subseteq F_1 \times \cdots \times F_n\), it contains the direct product of those halting configurations of the components which are also halting configurations of \(\Delta\).

We say that \(\Delta\) accepts the n-tuple \((\alpha_1, \ldots, \alpha_n)\), where \(\alpha_i, 1 \leq i \leq n\), is a sequence of multisets over \(O_i\) if the component \(\Pi_i\), starting from its initial configuration, using the symport/antiport rules as well as the inter-component communication rules in the non-deterministic maximally parallel way, takes from the environment the multiset sequence \(\alpha_i\), \(1 \leq i \leq n\), and \(\Delta\) eventually enters an accepting configuration.

We may associate languages to the dP automaton \(\Delta \equiv (O, \Pi_1, \ldots, \Pi_n, R, F)\), \(n \geq 1\), in several ways. The (concatenated) language of \(\Delta\) (introduced in [9]) over an alphabet \(\Sigma\) with respect to the mapping \(f = (f_1, \ldots, f_n)\) for \(f_i : \Sigma^* \rightarrow 2^{\Sigma^*}\), \(1 \leq i \leq n\), is defined as

\[L_{\text{concat}}(\Delta, f, \Sigma) = \{w_1 \ldots w_n \in \Sigma^* \mid w_i = f_i(v_{i,1}) \ldots f_i(v_{i,s_i})\} \text{ and } \alpha_i = v_{i,1} \ldots v_{i,s_i}, 1 \leq i \leq n, \text{ for an n-tuple of accepted multiset sequences } (\alpha_1, \ldots, \alpha_n)\}.

Now we introduce two variants of languages based on agreement of the components. The strong agreement language of \(\Delta\) over an alphabet \(\Sigma\) with respect to a mapping \(f = (g, \ldots, g)\) for \(g : \Sigma^* \rightarrow 2^{\Sigma^*}\), is defined as

\[L_{\text{agree}}(\Delta, f, \Sigma) = \{w \in \Sigma^* \mid w = g(v_1) \ldots g(v_s)\} \text{ and } \alpha = v_1 \ldots v_s, \text{ for an n-tuple of accepted multiset sequences } (\alpha, \ldots, \alpha)\] of \(\Delta\).

The weak agreement language of \(\Delta\) over an alphabet \(\Sigma\) with respect to a mapping \(f = (g, \ldots, g)\) for \(g : \Sigma^* \rightarrow 2^{\Sigma^*}\), is defined as

\[L_{\text{weak agree}}(\Delta, f, \Sigma) = \{w \in \Sigma^* \mid w = g(\alpha_i), 1 \leq i \leq n, \text{ for an n-tuple of accepted multiset sequences } (\alpha_1, \ldots, \alpha_n)\}.

that is, as opposed to the strong case, in weak agreement languages, it is not
necessary that $\alpha_i = \alpha_j$ for all $1 \leq i, j \leq n$, only the equality of $g(\alpha_i)$, the images
of $\alpha_i$, are required for all $1 \leq i \leq n$.

Notice also how the choice of $f$ essentially influences the power of the com-
ponents, and thus, the power of the whole dP automaton. In [9] the authors
consider dP automata with $f = (f', \ldots, f')$ where $f' : O^{(*)} \to 2^{O'}$ is defined in
such a way that a multiset over $O$ is mapped by $f'$ to the set of strings which
consists of all permutations of the elements of the multiset. In the following, we
will denote this mapping by $f_{\text{perm}}$.

In the case of $L_{\text{concat}}(\Delta, f, \Sigma)$, the words accepted by the components are
concatenated to obtain the words of the language accepted by the dP automata.
The language $L_{\text{s,agree}}(\Delta, f, \Sigma)$ contains the words which can be accepted by
all components, all of them reading (i.e., having as input) the same multiset
sequence during a successful computation, while the language $L_{\text{w,agree}}(\Delta, f, \Sigma)$
consists of all strings which can be accepted by all of the components simulta-
neously without requiring that the accepted multiset sequences are also the
same.

Finally, we recall a notion from [9]. A dP automaton $\Delta$ is called finite, if the
number of configurations reachable from its initial configuration is finite.

3 One-way multi-head finite automata versus finite
dP automata

In this section we describe the correspondence between the language family of
one-way finite multi-head automata and that of finite dP automata. We show
that the weak agreement language of any finite dP automaton with respect to
the mapping $f_{\text{perm}}$ is the language of a one-way multi-head automaton, and
moreover, the language of any one-way finite multi-head automaton can be ob-
tained as the strong or weak agreement language of a finite dP automaton with
respect to the mapping $f_{\text{perm}}$.

First we describe the weak agreement language of a finite dP automaton $\Delta$
with respect to the mapping $f_{\text{perm}}$.

**Theorem 1.** For any finite dP automaton $\Delta = (O, \Pi_1, \ldots, \Pi_k, R, F)$, $k \geq 2$, a
non-deterministic one-way $k$-head finite automaton $M = (Q, O, k, \delta, \triangleright, \triangleleft, q_0, F)$
can be constructed such that $L_{\text{w,agree}}(\Delta, f_{\text{perm}}, O) = L(M)$.

**Proof.** Let $\Delta = (O, \Pi_1, \ldots, \Pi_k, R, F)$, $k \geq 2$, be a finite dP automaton. We con-
struct the non-deterministic one-way $k$-head finite automaton $M = (Q, O, k, \delta, \triangleright,
\triangleleft, q_0, F)$ satisfying the conditions of the theorem such that the $i$-th head of $M$,
$1 \leq i \leq k$, will simulate the work of component $\Pi_i$ of $\Delta$. Namely, the strings
scanned by the $i$th head of $M$ are the strings accepted by component $\Pi_i$ of $\Delta$ with respect to the mapping $f_{\text{perm}}$. The equality of the two languages, i.e.,
$L_{\text{w,agree}}(\Delta, f_{\text{perm}}, O) = L(M)$, comes from the fact that the heads of $M$ read
the same word and the strings accepted by $\Delta$ are the strings which are accepted
by all components of $\Delta$. 

We start with an observation. Since $\Delta$ is a finite dP automaton, it has a finite number of different configurations $C_i$, $0 \leq i \leq s$. We denote the initial configuration by $C_0$, the set of configurations of $\Delta$ by $\mathcal{C}$. Since $\mathcal{C}$ is finite, the number of $k$-tuples of multisets $(u_1, \ldots, u_k)$ which can be imported from the environment by the skin membranes of the components in configuration $C_i$ to directly obtain configuration $C_j$, that is, where $(u_1, \ldots, u_k, C_i) \implies C_j$, should also be finite.

We construct the components of $M = (Q, O, k, \delta, >, <, q_0, F)$ as follows. The states of $\Delta$, which govern the computation, represent the transitions of $\Delta$. Let

$$Q = \{\langle C_i, C_j, (\alpha_1, \ldots, \alpha_k), r \rangle \mid C_i, C_j \in \mathcal{C}, (u_1, \ldots, u_k, C_i) \implies C_j, 0 \leq i, j \leq s, \alpha_j \in \text{perm}(u_j), 1 \leq j \leq k, 1 \leq r \leq 1 + \max\{|\alpha_l| \mid 1 \leq l \leq k\}\} \cup \{q_0\}.$$ 

Let us define some auxiliary notions. For any $q = \langle C_i, C_j, (\alpha_1, \ldots, \alpha_k), r \rangle \in Q - \{q_0\}$, where $0 \leq i, j \leq s$, $1 \leq r \leq \max\{|\alpha_l| \mid 1 \leq l \leq k\}$, let

$$\text{NextState}(q) = \{\langle C_i, C_j, (\alpha_1, \ldots, \alpha_k), r + 1 \rangle\},$$

and for $r = 1 + \max\{|\alpha_l| \mid 1 \leq l \leq k\}$, let

$$\text{NextState}(q) = \{\langle C_j, C_i, (\beta_1, \ldots, \beta_k), 1 \rangle \in Q\}.$$ 

Let also $\text{NextState}(q_0) = \{\langle C_0, C_1, (\alpha_1, \ldots, \alpha_k), 1 \rangle \in Q\}$, and moreover, let $F = \{\langle C_i, C_j, (\alpha_1, \ldots, \alpha_k), r \rangle \in Q \mid C_j \in F, r = 1 + \max\{|\alpha_l| \mid 1 \leq l \leq k\}\}$. 

Now, we define the transition rules in $\delta$. Any transition in $\Delta$ corresponds to a set of transitions sequences in $M$, where the number of steps performed by $M$ is equal to the maximum of the number of objects imported by the components in $\Delta$ in the simulated transition.

Let $\delta$ consist of transition rules $q' \in \delta(q, \alpha_1, \ldots, \alpha_k)$ for $q' \in \text{NextState}(q)$, $q = \langle C_i, C_j, (\alpha_1, \ldots, \alpha_k), r \rangle \in Q$, and $a_j = \alpha_j[r]$ for $1 \leq r \leq |\alpha_j|$, or $a_j = \lambda$ for $|\alpha_j| + 1 \leq r \leq 1 + \max\{|\alpha_l| \mid 1 \leq l \leq k\}$ (where $\alpha_j[r]$ denotes the $r$th symbol of the string $\alpha_j$ for some $1 \leq r \leq |\alpha_j|$).

Now we show that any string $w$ in $L_{w, \text{agree}}(\Delta, \text{perm}, O)$ can be accepted by $M$. If $w \in L_{w, \text{agree}}(\Delta, \text{perm}, O)$, then there exists a computation $c_w$ in $\Delta$ where $c_w = C_0, C_1^{(w)}, \ldots, C_h^{(w)}$, $h \geq 0$, with $C_h^{(w)} \in F$ such that $w = x_1^{(i,w)} \ldots x_h^{(i,w)}$, where $x_j^{(i,w)} \in \text{perm}(u_j^{(i,w)}), 1 \leq i \leq k, 1 \leq j \leq h$, and $u_j^{(i,w)}$ is the multiset of objects imported from the environment by component $i$ at step $j$ of computation $c_w$. Notice that $x_1^{(i,w)} \ldots x_h^{(i,w)}$ is the same string for all $i$, $1 \leq i \leq k$.

It is easy to see that then there exists a corresponding accepting computation $c_w'$ of $w$ in $M$. At the beginning of the computation $M$ is in state $q_0$. Let $x_j^{(i,w)} = a_{1,j}^{(i,w)} \ldots a_{m_j,j}^{(i,w)}$, where $a_{1,j}^{(i,w)}, \ldots, a_{m_j,j}^{(i,w)} \in O, 1 \leq j \leq h, m_j \geq 1, 1 \leq i \leq k$.

Since $x_j^{(i,w)} \in \text{perm}(u_j^{(i,w)}), 1 \leq i \leq k, 1 \leq j \leq h$, $M$ has transition rules of the form $q' \in \delta(q, b_1, \ldots, b_k)$, where $q = \langle C_j, C_{j+1}, (x_{j,1}^{(i,w)}), \ldots, x_{j,h}^{(i,w)}, r \rangle$, with $(u_j^{(1,w)}, \ldots, u_j^{(k,w)}, C_j) \implies C_{j+1}, 1 \leq j \leq h - 1$, and $c_w = C_0, C_1, \ldots, C_j, C_{j+1}$,
\[ \ldots, C_h. \] Furthermore, \( b_i = a^{(i,w)}_{r,j}, 1 \leq i \leq k, \) for \( 1 \leq r \leq \max\{|x^{(i,w)}_{j}| \mid 1 \leq i \leq k\} \) and \( b_i = \lambda \) for \( |x^{(i,w)}_{j}| < r \leq 1 + \max\{|x^{(i,w)}_{j}| \mid 1 \leq i \leq k\} \).

We explain how \( M \) simulates \( \Delta \). Each of the \( k \) heads reads \( w \) in \( h \) blocks of computation steps, namely, the \( i \)th head, \( 1 \leq i \leq k, \) reads \( x^{(i,w)}_1, \ldots, x^{(i,w)}_h \) where \( w = x^{(i,w)}_1 \ldots x^{(i,w)}_h \). The heads start and finish the reading of the corresponding subword \( x^{(i,w)}_j, 1 \leq i \leq k, 1 \leq j \leq h \) at the same time, they read their own substrings in a synchronized manner as follows: The head makes no move, i.e., it reads \( \lambda \) if the length of its subword is less than the maximum of the lengths of these strings. Due to the definition of the accepting state of \( M, \) \( L \) corresponds to the set of accepting configurations of \( \Delta \). Since the heads of \( M \) read the same string, it is easy to see that any string in \( L_{w,\text{agree}}(\Delta, f_{\text{perm}}, O) \) can be accepted by \( M \). The reverse statement also holds. Due to the construction of the transition function \( \delta \) of \( M \), all the strings \( w = x^{(i,w)}_1 \ldots x^{(i,w)}_h \) and no more words are accepted by \( M \), where \( x^{(i,w)}_j, 1 \leq i \leq k, 1 \leq j \leq h \) is a permutation of the multiset of objects \( u^{(i,w)}_j \) imported from the environment by component \( i \) at step \( j \) of an accepting computation \( c_w \) in \( \Delta \) such that the components accept the same set of strings. This implies that \( L_{w,\text{agree}}(\Delta, f_{\text{perm}}, O) = L(M) \) holds.

Analyzing the above proof, the reader may observe that for any finite dP automaton \( \Delta' \) where in the course of (accepting) computations each component automaton imports from the environment at most one object in each computational step (i.e., either the empty multiset or a multiset consisting of a single object), it holds that with respect to the mapping \( f_{\text{perm}} \), the weak agreement language coincides with the strong agreement language. Thus, in this case the construction in Theorem 1 yields a multi-head finite automaton \( M' \) with \( L(M') = L_{w,\text{agree}}(\Delta', f_{\text{perm}}, O) = L_{s,\text{agree}}(\Delta', f_{\text{perm}}, O) \).

Now we prove the converse statement as follows.

**Theorem 2.** For any non-deterministic one-way \( k \)-head finite automaton \( M, \) \( k \geq 2, \) with input alphabet \( \Sigma \) we can construct a finite dP automaton \( \Delta \) of degree \( k \), such that

\[ L(M) = L_{s,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma) = L_{w,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma). \]

**Proof.** Let \( M = (Q, \Sigma, k, \delta, \mathcal{O}, q_0, F), k \geq 2, \) be a non-deterministic one-way \( k \)-head finite automaton. To prove the statement, we first construct a finite dP automaton \( \Delta = (O, \Pi_1, \ldots, \Pi_k, R, \mathcal{F}), \) such that \( L(M) = L_{s,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma) \) holds. The idea of the proof is that the transitions in \( M \) are simulated by \( \Delta \) in a cycle, the work of each reading head of \( M \) is executed by a different component.

Let \( I = \{[q, x_1, \ldots, x_k] \mid q \in Q, x_i \in \Sigma \cup \{\Lambda\}\}, \) and let \( O = \Sigma \cup \{S, X_1\} \cup I \cup \{X_i^\alpha \mid \alpha \in I, 1 \leq i \leq k\}. \) The symbols of \( I \) govern the work of \( \Delta \) by storing the actual state of \( M \) and the letters from \( \Sigma \) that the heads read during the next transition which is simulated by \( \Delta \) in a cycle, above.
Let us define the set of communication rules $R$ of $\Delta$ as

$$R = \{(s_i, X_\alpha^a/s_{i+1}, X_\alpha^{i+1}), (s_i, xX_\alpha^{i+1}/s_{i+1}, X_\alpha^i) \mid x \in \Sigma, \alpha = [q, x_1, \ldots, x_k] \in I, 1 \leq i \leq k - 1\} \cup \{(s_k, X_k^\alpha/\alpha, \lambda), (s_k, X_k^\alpha/x/s_1, X_1), (s_k, X_1/s_1, X_k^\alpha) \mid x \in \Sigma, \alpha = [q, x_1, \ldots, x_k] \in I\}. $$

(Recall that $s_i$, $1 \leq i \leq k$, denotes the skin membrane of component $H_i$.)

We define $H_i = (\Omega, ([ \ ]_{s_i}, P_{i,1}, P_{i,2},([ \ ]_{s_i}, w_{i,1}, w_{i,2}), \mathcal{F}_i)$ with

$$w_{i,1} = aS_{X_1} \cup \bigcup_{\alpha \in I} (X_1^\alpha), \text{ where } a \notin \Sigma,$$

$$P_{i,1} = \{(a, out; x_1, in)_{\alpha} \mid \alpha = [q, x_1, \ldots, x_k], x_1 \neq \lambda\},$$

$$P_{i,2} = I \cup \bigcup_{\alpha \in I} (X_k^\alpha),$$

$$P_{i,2} = \{(a, out; S, in) \mid \alpha = [q_0, x_1, \ldots, x_k] \in I\} \cup \{(X_k^\beta, out; X_k^\alpha, in) \mid \alpha = [q, x_1, \ldots, x_k], \beta = [s, y_1, \ldots, y_k], \text{ such that there is } s \in \delta(q, x_1, \ldots, x_k)\},$$

$$\mathcal{F}_i = O^{(i)} \times \{S \cup I \cup (\bigcup_{\alpha \in I} X_k^\alpha) - \beta \mid \beta = [q_f, x_1, \ldots, x_k], q_f \in F\}.$$

The first component is responsible for launching the simulation of a transition in $M$, for simulating the movement of the first head and also for finishing the computation. The other components simulate the movements of the other heads of $M$.

Let $H_i = (\Omega, ([ \ ]_{s_i}, P_i, ([ \ ]_{s_i}, w_i), \mathcal{F}_i)$ for $2 \leq i \leq k$ be defined as follows.

$$w_i = \bigcup_{\alpha \in I} (X_i^\alpha),$$

$$P_i = \{(x, out; x_i, in)_{\alpha} \mid \alpha = [q, x_1, \ldots, x_k] \in I, x_i \neq \lambda, x_i \in \Sigma\},$$

$$\mathcal{F}_i = O^{(i)}.$$

Now we explain the work of $\Delta$ in more details. The work of $\Delta$ starts in the initial configuration $((\mu_1, aS_{X_1}u_1, u_1), (\mu_2, u_2), \ldots, (\mu_n, u_k))$ where $u_i$ consists of all symbols $X_\alpha^a$ for all $\alpha \in I, 1 \leq i \leq k$, and $u_{1,2} = I \cup \{X_k^\alpha \mid \alpha \in I\}$. The next configuration is

$$((\mu_1, a \alpha X_1u_1, u_{1,2}S), (\mu_2, u_2), \ldots, (\mu_n, u_k))$$

for some $\alpha = [q_0, x_1, \ldots, x_k]$ and $u_{1,2}'$ containing all symbols of $u_{1,2}$ except $\alpha$. Now, if $x_1 \neq \lambda$, then it is imported from the environment by the skin region of the first component while $X_\alpha^a$ is exchanged with $X_\alpha^2$ by applying a communication rule from $R$. We obtain a configuration

$$((\mu_1, x'_1X_2^\alpha X_1u'_1, u_{1,2}S), (\mu_2, \alpha X_1^a u'_2), \ldots, (\mu_n, u_k))$$
where if $x_1 \neq \lambda$ then $x_1' = x_1$, otherwise $x_1' = a$. The multisets $u_1'$ and $u_2'$ contain the symbols from $u_1$ and $u_2$ with the exception of $X_1^\alpha$ and $X_2^\alpha$, respectively. In the next step $X_1^\alpha$ and $X_2^\alpha$ are exchanged one more time and $x_1'$ is transferred to the second component, thus we obtain
\[
((\mu_1, X_1^\alpha u_1, u_1', \alpha X_2^\alpha u_2'), \ldots, (\mu_k, u_k))
\]
by communication between the skin membranes.

Now we discuss the cases $k = 2$ and $k \geq 3$ separately. Suppose that $k \geq 3$.

Then the next configuration is
\[
((\mu_1, X_1 u_1, u_1', \alpha X_2 u_2'), (\mu_2, x_2' \alpha X_2^\alpha u_2'), \ldots, (\mu_k, u_k))
\]
where $u_1 = X_1^\alpha u'$ and $x_2' = x_2 \neq \lambda$, or if $x_2 = \lambda$, then $x_2' = x_1'$.

The system now works analogously to the steps described above by transferring $\alpha$ to the $(i + 1)$th component while reading $x_i$ from $\alpha = [q_0, x_1, \ldots, x_k]$, for $2 \leq i \leq k - 1$, reaching the configuration
\[
((\mu_1, X_1 u_1, \alpha X_2^\alpha u_1', \alpha X_2 u_2'), \ldots, (\mu_k, x_k' \alpha X_2^\alpha u_k'))
\]
where $x_k' = x_k - 1 \neq \lambda$, or if $x_k = \lambda$, then $x_k' = x_k - 2$.

Now the symbol $x_k$ is read by the $k$th component and it also communicates with the first one obtaining
\[
((\mu_1, X_1 \alpha X_2^\alpha u_1', \alpha X_2^\alpha u_1'), (\mu_2, u_2') \ldots, (\mu_k, x_k' X_1 u_k'))
\]
where $u_1'$ is the multiset $u_1$ without the symbol $X_1^\alpha$, and $x_k' = x_k \neq \lambda$, or if $x_k = \lambda$, then $x_k' = x_k - 1$. Now, with the help of the symbol $X_1$, $x_k'$ is also transferred to the first component, and the exchange of symbols between the skin and the inner regions of the first component results in
\[
((\mu_1, x_k' \beta X_1^\alpha u_1, \alpha X_2^\alpha u_1'), (\mu_2, u_2'), \ldots, (\mu_k, x_k' X_1 u_k'))
\]
where $u_1'$ is the same multiset as above, and $u_1''$ is obtained from $u_1$ by deleting symbol $\beta$ which is a new transition symbol such that $\beta = [q, y_1, \ldots, y_k]$ for some $q, y_i \in \delta(q_0, x_1, \ldots, x_k)$. Now $X_1^\alpha$ and $X_1$ are exchanged again, and the reading of the symbols $y_i$, $1 \leq i \leq k$, will follow in the same way as described above for $x_1, \ldots, x_k$.

For $k = 2$, after entering configuration $((\mu_1, X_1^\alpha u_1, u_1', \alpha X_2^\alpha u_2'), (\mu_2, x_2' \alpha X_2^\alpha u_2'))$, multiset $x_1'$ is changed for $x_2$ and multiset $\alpha X_2^\alpha$ is exchanged by communication with $X_1^\alpha$. Then the first component changes $\alpha X_2^\alpha$ for $\beta X_2^\alpha$, and the second component $x_1'$ for $x_2$. In the next step $X_2^\alpha$ at the first component is exchanged with $X_1$ at the second component, thus the components will have a configuration in which the simulation of a transition of $M$ may start.

This way, each component of $\Delta$ continues to simulate one of the reading heads of $M$. The process may end if $\Delta$ reaches an accepting configuration, that is, if a transition symbol $[q_f, x_1, \ldots, x_k]$ with $q_f \in F$ is imported from the second region of the first component into the first region.
Since in each step of the computation of $\Delta$ at most one symbol from $\Sigma \subset O$ is read from the environment, the mapping $f_{\text{perm}}$ transforms any input multiset sequence $a_1 \ldots a_t \in (\Sigma^*)^*$ into the unique string $a_1 \ldots a_t \in \Sigma^*$, thus, the strong agreement language $L_{s,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma)$ coincides with the language $L(M)$ accepted by the multi-head finite automaton $M$. For the same reasons, i.e., since in each step of the computation of $\Delta$ at most one symbol from $\Sigma \subset O$ is read from the environment, $L_{w,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma) = L(M)$ also holds.

In [5] it was shown that any regular language, i.e., any language accepted by a (one-way one-head) finite automaton is equal to the language of a finite $P$ automaton with respect to the mapping $f_{\text{perm}}$. Since a $P$ automaton can be considered as a $dP$ automaton with one component, the statement above also holds for $k = 1$, that is, for any finite automaton $M$ over the input alphabet $\Sigma$, we can construct a $dP$ automaton $\Delta$ with $L(M) = L_{s,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma) = L_{w,\text{agree}}(\Delta, f_{\text{perm}}, \Sigma)$.

4 Two-way multi-head finite automata versus finite dP automata

In this section we introduce the notion of a two-way $dP$ automaton and we show how two-way finite $dP$ automata characterize the language family accepted by non-deterministic two-way multi-head finite automata.

We first recall that alphabets of the form $\Sigma \cup \tilde{\Sigma}$, where $\Sigma$ is an alphabet itself and $\tilde{\Sigma} = \{ \tilde{a} \mid a \in \Sigma \}$ are called double alphabets [1].

We first start with some discussion on the movement of the heads of a two-way multi-head finite automaton in the course of computing.

Let $w \in (\Sigma \cup \tilde{\Sigma})^*$ be a word over a double alphabet $\Sigma \cup \tilde{\Sigma}$. We say that $w$ is a two-way trail over $\Sigma \cup \tilde{\Sigma}$ if it can be written as

$$w = w_1\tilde{x}_1\tilde{w}_2x_2\tilde{w}_3\tilde{x}_3\tilde{w}_4x_4 \ldots \tilde{x}_{n-2}\tilde{w}_{n-1}x_{n-1}w_n,$$

for $x_i \in \Sigma$, $\tilde{x}_i \in \tilde{\Sigma}$, $\tilde{w}_i \in \tilde{\Sigma}^*$, $1 \leq i \leq n$, such that the following properties are satisfied.

1. For all $w'$ with $w = w'w''$, it holds that $|w'|_\Sigma \geq |w''|_\Sigma$;
2. for all $w''$ with $w = w'w''$, it holds that $|w'|_\Sigma \geq 2 \cdot |w''|_\Sigma$;
3. for all $1 \leq i \leq n - 2$, the subwords $w_i\tilde{x}_i\tilde{w}_{i+1}$ are of the form $w_i'\tilde{x}_i\tilde{w}_{i+1}'$ where $w_i = w_i'x_i$, $\tilde{w}_{i+1} = \tilde{x}_i\tilde{w}_{i+1}'$, and the subwords $\tilde{w}_{i+1}x_{i+1}w_{i+2}'$ are of the form $\tilde{w}_{i+1}'\tilde{x}_{i+1}x_{i+2}'$ where $\tilde{w}_{i+1} = \tilde{w}_{i+1}'\tilde{x}_{i+1}$, $w_{i+2} = x_{i+2}'$;
4. for all $1 \leq i \leq n - 2$, the subwords $w_i\tilde{x}_i\tilde{w}_{i+1}$ satisfy one of the properties (a) $\tilde{w}_{i+1} = w_i^R\tilde{w}_{i+1}'$, or (b) $w_i = w_i^R\tilde{w}_{i+1}'$, while the subwords $\tilde{w}_{i+1}x_{i+1}w_{i+2}'$ satisfy one of the properties (c) $w_{i+2} = w_{i+2}'\tilde{w}_{i+2}'$, or (d) $\tilde{w}_{i+1} = w_{i+1}^R\tilde{w}_{i+2}'$, depending on the length of $w_i$, $w_{i+1}$, and $w_{i+2}$.

Observe that these conditions describe the two-way movement of a reading head of a two-way finite automaton over a finite string $u \in \Sigma^*$: Letters $x \in \Sigma$
which are read in such a way that the head moves right after reading them are marked with \( x \), letters \( x \in \Sigma \) which are read in such a way that the head moves left afterward are marked with \( \bar{x} \), and moreover, the head starts reading on the leftmost symbol and finishes reading with the rightmost symbol.

Property 1. describes that the head never moves to the left of the left endmarker, i.e., it performs at least as many moves to the right, as moves to the left. Property 2. guarantees that the head finishes its movement on the last symbol, i.e., if it turns to the left and performs a number of moves to the left, then it should also perform at least the same number or moves to the right. Property 3 identifies the subwords which describe the turn of the direction, and property 4 requires that after turning, the same symbols are read in the inverse order as the ones that were read before the turn.

We note that a similar approach can be found in [1], the difference of the two formalisms arises from the different notations used in the definition of a two-way automaton. We first note that the concept of a two-way trail can be extended to the concept of a two-way finite automaton.

Using the notion of the two-way trail, it is easy to see that a computation in a 2NFA\((k)\) \( M = (Q, \Sigma, k, \delta, \nu, q_0, F) \) is accepting if and only if it is given by a sequence states \( q_0, q_1, \ldots, q_n, q_s = q_f \in F \), where \( q_j \in \delta(q_{j-1}, x_{j-1,1}, \ldots, x_{j-1,k}) \), 1 \( \leq j \leq s - 1 \) such that \( w = x_{1,1} \ldots x_{s,1} \in (\Sigma \cup \bar{\Sigma})^* \) is a two-way trail and \( x_{j,i} \in (\Sigma \cup \bar{\Sigma} \cup \{\lambda\}) \) for \( 1 \leq j \leq s, 1 \leq i \leq k \).

On the analogy of 2NFA\((k)\), we will define the concept of a two-way finite dP automaton. We first note that the concept of a two-way trail can be extended to the concept of a two-way multiset trail in an obvious manner. Namely, a sequence of multisets

\[
u = u_1 \bar{u}_1 u_2 x_2 u_3 \bar{u}_3 x_4 \ldots \bar{u}_{n-2} u_{n-2} x_{n-2} u_{n-1} x_{n-1} u_n,
\]
defined over a double alphabet \( O \cup \bar{O} \), where \( x_i \in O, u_i \in (O^{(\bar{\nu})})^*, \bar{x}_i \in \bar{O}, \bar{u}_i \in (\bar{O}^{\nu})^*, 1 \leq i \leq n, \) is a two-way multiset trail if \( u \) satisfies properties 1-4, given above.

A dP automaton \( \Delta = (\Pi_1, \ldots, \Pi_k, R, F) \) where \( O' = O \cup \bar{O} \) is a double alphabet is called a two-way dP automaton if any multiset \( u_i \) which enters component \( \Pi_i, 1 \leq i \leq k \), in the course of a computation consists of either objects of \( O \), or objects of \( \bar{O} \), or it is the empty multiset.

Obviously, if a two-way dP automaton is a finite dP automaton, then we speak of a two-way finite dP automaton.

In the following we define the strong agreement language and the weak agreement language of a two-way dP automaton \( \Delta \).

We first define a reduction mapping for two-way trails.

Let \( h : (\Sigma \cup \bar{\Sigma})^* \rightarrow 2^{(\Sigma \cup \bar{\Sigma})^*} \) be defined as follows:

- \( w_1 \bar{x}_1 w_2 \in h(w) \) if and only if \( w = w_1 x_1 \bar{x}_1 x_2 w_2 \), for some \( x_1, x_2 \in \Sigma \), or
- \( w_1 x_1 w_2 \in h(w) \) if and only if \( w = w_1 \bar{x}_1 x_2 x_1 w_2 \), for some \( x_1, x_2 \in \Sigma \), or
- \( h(w) = w \) if and only if \( w \in \Sigma^* \), or
- let \( h(w) \) be undefined otherwise.
Let us define \( h^0(w) = h(w) \), \( h^i(w) = h(h^{i-1}(w)) \) for \( i \geq 0 \), and let \( h^*(w) = h^i(w) \) for some \( i \), such that \( h^i(w) = h^{i+1}(w) \).

Analogously to the method used in [1], it can be shown that \( w \) is a two-way trail (called in [1] a trace of a two-way computation) if and only if \( h^*(w) \) is a unique word. The proof follows from the (diamond) Church-Rosser property of the operation induced by mapping \( h \).

Now, if mapping \( h^* \) is applied to a two-way trail \( w \in (\Sigma \cup \bar{\Sigma})^* \), then \( h^*(w) \in \Sigma^* \) is the word which was written on the tape of the two-way reading head producing the two-way trail \( w \). In the following, for a two-way trail \( w \) we call \( h^*(w) \) the \( h \)-reduction of \( w \).

The definitions of mapping \( h \) and the \( h \)-reduction of a two-way trail can be extended to mapping \( h_m \) and to \( h_m \)-reduction for two-way multiset trails with the obvious modifications, using again the fact that a two-way multiset trail is a sequence of multisets satisfying certain properties.

As in the case of finite dP automata over arbitrary alphabets, we define the strong agreement language and the weak agreement language of a two-way dP automaton.

The strong agreement language \( L_{s,\text{agree}}(\Delta, f, \Sigma) \) of a two-way dP automaton \( \Delta = (O', \Pi_1, \ldots, \Pi_k, R, F) \) over an alphabet \( \Sigma \) with respect to a mapping \( f = (g, \ldots, g) \) for \( g : O^{(s)} \rightarrow 2^{\Sigma^*} \) is defined as

\[
L_{s,\text{agree}}(\Delta, f, \Sigma) = \{ w \in \Sigma^* \mid w = g(v_1) \ldots g(v_s), \alpha = v_1 \ldots v_s, \text{ where } v_j \in O^{(s)}, 1 \leq j \leq s, \text{ and } \alpha = h_m^*(\beta_i), \text{ for a } k\text{-tuple of accepted multiset sequences } (\beta_1, \ldots, \beta_k), \text{ where } \beta_i \text{ is a two-way multiset trail, } 1 \leq i \leq k \}.
\]

Thus, words of the strong agreement language of \( \Delta \) over \( \Sigma \) with respect to \( f \) are obtained as follows: we consider all \( k \)-tuples of multiset sequences accepted by \( \Delta \) where any multiset sequence is a two-way multiset trail and its \( h_m \)-reduction is the same multiset sequence. Then, we consider the \( f \)-images of the \( h_m \)-reductions of these two-way multiset trails.

The weak agreement language \( L_{w,\text{agree}}(\Delta, f, \Sigma) \) of a two-way dP automaton \( \Delta \) over an alphabet \( \Sigma \) with respect to a mapping \( f = (g, \ldots, g) \) for \( g : O^{(s)} \rightarrow 2^{(\Sigma \cup \bar{\Sigma})^*} \), is defined as

\[
L_{w,\text{agree}}(\Delta, f, \Sigma) = \{ w \in \Sigma^* \mid w = h^*(u_i), 1 \leq i \leq k, u_i = g(v_{1,i}) \ldots g(v_{s_{i-1},i}), \text{ is a two-way trail, } \alpha_i = v_{1,i} \ldots v_{s_{i-1},i}, \text{ for } s_i \geq 1, (\alpha_1, \ldots, \alpha_k) \text{ is a } k\text{-tuple of multiset sequences accepted by } \Delta \}.
\]

The words of the weak agreement language of \( \Delta \) over an alphabet \( \Sigma \) and with respect to mapping \( f \) are obtained as follows: we consider \( f \)-images of \( k \)-tuples of multiset sequences accepted by \( \Delta \). If they are two-way trails and their \( h \)-reduction is the same string, then this string belongs to the weak agreement language of \( \Delta \) over \( \Sigma \) and with respect to mapping \( f \).
Now, as two-way trails correspond to the movement of the reading heads of two-way automata (not necessarily finite), the two-way trails read by a finite dP automaton with $k$ components correspond to the computations of a 2NFA($k$). Using similar arguments as used for the one-way case in Theorem 2 and Theorem 1, we can also construct two-way finite automata and two-way finite dP automata which accept the same languages.

**Theorem 3.** Any language that can be accepted by a non-deterministic two-way $k$-head finite automaton for $k \geq 2$ is equal to the strong or weak agreement language of a two-way finite dP automaton of degree $k$ with respect to the mapping $f_{\text{perm}}$.

**Proof.** Let $M' = (Q, \Sigma, k, \delta, \tau, o, q_0, F)$, $k \geq 2$, be a non-deterministic two-way $k$-head finite automaton. We construct a two-way finite dP automaton $\Delta' = (O \cup \bar{O}, \Pi'_1, \ldots, \Pi'_k, R', F)$, such that $L(M') = L_{\text{agree}}(\Delta', f_{\text{perm}}, \Sigma)$ holds. To do this, we slightly modify the construction that can be found in the proof of Theorem 2; this is the reason why we use notations $M'$ and $\Delta'$ as well. Firstly, we consider the double alphabet $\Sigma \cup \bar{\Sigma}$. Then we modify $I = \{[q, x_1, \ldots, x_k] \mid q \in Q, x_i \in \Sigma \cup \{\lambda\}\}$ (from the proof of Theorem 2) to $I'$ such that in every $[q, x_1, \ldots, x_k] \in I$ symbol $x_i$ is in $\bar{\Sigma}$ if after reading $x_i$ the $i$th head of $M'$ moves to the right and it is replaced by $\bar{x}_i$ if after reading this letter the head of $M'$ moves to the left. We then consider the double alphabet $O' \cup \bar{O}'$, where $O'$ consists of the barred versions of symbols in $O$ (notice that the only objects we really need are elements of $\Sigma$), where $O' = \Sigma \cup \{X_1\} \cup I' \cup \{X_\alpha^\pi \mid \alpha \in I', 1 \leq i \leq k\}$. Then, we perform the necessary changes in the components of $\Delta$ in the proof of Theorem 2 accordingly, i.e., we replace $O$ by $O'$ and $I$ by $I'$. It can easily be seen that the components of $\Delta'$ work in an analogous manner to the components of $\Delta$. If a $k$-tuple of sequences of multisets accepted by $\Delta'$ is a $k$-tuple of two-way multiset trails, then $\Delta'$ simulates the behavior of $M'$, and this implies that the statement for the strong agreement language of $\Delta'$ holds. Since any component of $\Delta'$ imports at each step of an accepting computation only one object (one symbol) from the environment, and thus $f_{\text{perm}}$ assigns one letter to the imported multiset, the statement concerning the weak agreement language holds as well.

**Theorem 4.** Any language which is the weak agreement language with respect to the mapping $f_{\text{perm}}$ of a two-way finite dP automaton of degree $k$, $k \geq 2$, can be accepted by a non-deterministic two-way $k$-head finite automaton.

**Proof.** The proof can be obtained from the construction of the proof of Theorem 1 in an obvious manner. Instead of $\Delta = (O, \Pi_1, \ldots, \Pi_k, R, F)$, $k \geq 2$, we consider $\Delta' = (O', \Pi'_1, \ldots, \Pi'_k, R', F')$, where $O' = O \cup \bar{O}$ is a double alphabet. Then we construct a $k$-head automaton $M'$ in the same way as $M$ is constructed in that proof. Now if we consider the set of $k$-tuples of strings which are $f_{\text{perm}}$-images of the $k$-tuples of multiset sequences accepted by $\Delta$ and where the strings are two-way trails, then we obtain the set of $k$-tuples of strings describing the movements of the $k$ heads of $M'$, a non-deterministic two-way $k$-head finite automaton. Then we take the $h$-reductions of the strings in these $k$-tuples. If they are equal, then
the $k$-tuple of strings provides a word in the language of $M'$ which is equal to the weak agreement language of $\Delta'$ with respect to $f_{perm}$ and over $O'$.

5 Further remarks

In the previous sections we demonstrated how languages of non-deterministic finite multi-head automata can be obtained by finite dP automata. Since the theory of multi-head finite automata is a highly elaborated field, the obtained results have several consequences. Since $\text{NSPACE}(\log n) = \bigcup_{k \geq 1} \mathcal{L}(2\text{NFA}(k))$ [6], Theorems 4 and 3 provide characterizations of this important complexity class in terms of finite dP automata. Another important result is (for details, see [7] and the papers cited in the article), that the emptiness, finiteness, infiniteness, universality, inclusion, equivalence, regularity and context-freeness are not semidecidable for $\mathcal{L}(1\text{NFA}(k))$ and $\mathcal{L}(2\text{NFA}(k))$, $k \geq 2$. This implies that these properties hold for certain variants of language classes of finite dP automata as well. We also note that the terms determinism, head reversal, sensing head, stateless automaton, synchronized moving of heads, data-independence which are known and studied in detail for multi-head finite automaton, can be interpreted and implemented for finite dP automata in a natural manner. We plan research in these directions in the future. Similarly, dP automata are suitable tools for describing multi-head pushdown automata or other variants of automata with more than one reading heads. Investigations of these constructs are also among our objectives in the future.

References


An Adaptive Algorithm for P System Synchronization

Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
{mjd,yun,radu}@cs.auckland.ac.nz

Abstract. We present an improved solution for the Firing Squad Synchronization Problem (FSSP) for digraph-based P systems. We improve our previous FSSP algorithm by allowing the general to delegate a more central cell in the P system to send the final command to synchronize. With $e$ being the eccentricity of the general and $r$ denoting the radius of the underlying digraph, our new algorithm guarantees to synchronize all cells of the system, between $e + 2r + 3$ steps (for all trees structures and many digraphs) and up to $3e + 7$ steps, in the worst case for any digraph. Empirical results show our new algorithm for tree-based P systems yields at least 20% reduction in the number of steps needed to synchronize over the previous best-known algorithm.

Keywords: cellular automata, firing squad synchronization, P systems.

1 Introduction

The Firing Squad Synchronization Problem (FSSP) is one of the best studied problems for cellular automata, originally proposed by Myhill in 1957 [12]. The initial problem involves finding a cellular automaton, such that after the “firing” order is given by the general, after some finite time, all the cells in a line enter a designated firing state, simultaneously and for the first time. For an array of length $n$ with the general at one end, minimal time $(2n - 2)$ solutions was presented by Goto [6], Waksman [20] and Balzer [2]. Several variations of the FSSP have been proposed and studied [13, 17]. The FSSP have been proposed and studied for variety of structures [10, 14, 7, 4].

The synchronization problem has recently been studied in the framework of P systems, specifically for tree-based P systems. Initially, Bernardini et al. [3] provided a deterministic solution, based on a depth-first search, with time complexity $4n + 2h$, for transitional P systems with priorities and polarizations, where $h$ is the height of the tree structure underlying the P system and $n$ is the number of membranes of the P system. Later, Alhazov et al. [1] provided an improved deterministic algorithm that runs in $3h + 3$ steps, for transitional P systems with promoters and inhibitors. Their solution uses a breadth-first search (BFS), a broadcast and a convergecast.
In this paper, we present an improved FSSP solution for tree-based P systems with states and priorities, where the key improvement comes in having the general delegate a more central cell, as an alternative to itself, to broadcast the final “firing” order, to enter the firing state. This solution uses BFS, broadcast and convergecast operations. We also give details on how to use this approach to improve the synchronization time of digraph-based P systems with states, priorities and promoters.

It is well known in cellular automata [19], where “signals” with propagating speeds 1/1 and 1/3 are used to find a half point of one-dimensional arrays; the signal with speed 1/2 is reflected and meets the signal with speed 1/3 at half point. We generalize the idea used in cellular automata to find the center of a tree that defines the membrane structure of a P system.

Let $r$ denote the radius of the underlying graph of a digraph, where $e/2 \leq r \leq e$. Our new algorithm is guaranteed to synchronize in $t$ steps, where $e + 2r + 3 \leq t \leq 3e + 7$. In fact, the lower bound is achieved, for all digraphs that are trees.

In addition to our FSSP solution, determining a center cell has many potential real world applications, such as facility location problems and broadcasting.

The rest of the paper is organized as follows. In Section 2, we give some basic preliminary definitions including our P system model and formally introduce the synchronization problem that we solve. In Section 3, we provide a detailed P system specification for solving the FSSP for tree-based P systems. In Section 4, we provide a detailed P system specification for solving the FSSP for digraph-based P systems. Finally, in Section 5, we summarize our results and conclude with some open problems.

2 Preliminaries

We assume that the reader is familiar with the basic terminology and notations, such as relations, graphs, nodes (vertices), edges, directed graphs (digraphs), directed acyclic graphs (dag), arcs, alphabets, strings and multisets.

For a digraph $(X, \delta)$, recall that $\text{Neighbor}(x) = \delta(x) \cup \delta^{-1}(x)$. The relation Neighbor is always symmetric and defines a graph structure, which will be here called the virtual communication graph defined by $\delta$.

A special node $g \in X$ is designated as the general. For a given general $g$, we define the depth of a node $x$, $\text{depth}_g(x) \in \mathbb{N}$, as the length of a shortest path between $g$ and $x$, over the Neighbor relation. Recall that the eccentricity of a node $x \in X$, $\text{ecc}(x)$, as the maximum length of a shortest path between $x$ and any other node. We note $\text{ecc}(g) = \max\{\text{depth}_g(x) \mid x \in X\}$.

Recall that a (free or unrooted) tree has either one or two center nodes—any node with minimum eccentricity. We denote a tree $T = (X, A)$, rooted at node $g \in X$ by $T_g$. The height of a node $x$ in $T_g$ is denoted by $\text{height}_g(x)$. For a tree $T_g$, we define the middle node to be the center node closest to $g$ of the underlying tree $T$ of $T_g$. Let $T_g(x)$ denote the subtree rooted at node $x$ in $T_g$.

Given nodes $x$ and $y$, if $y \in \text{Neighbor}(x)$ and $\text{depth}_g(y) = \text{depth}_g(x) + 1$, then $x$ is a predecessor of $y$ and $y$ is a successor of $x$. Similarly, a node $z$ is a peer...
An Adaptive Algorithm for P System Synchronization

of $x$, if $z \in \text{Neighbor}(x)$ and $\text{depth}_g(z) = \text{depth}_g(x)$. Note that, for node $x$, the set of peers and the set of successors are disjoint with respect to $g$. For node $x$, $\text{Pred}_g(x) = \{y \mid y$ is a predecessor of $x\}$, $\text{Peer}_g(x) = \{y \mid y$ is a peer of $x\}$ and $\text{Succ}_g(x) = \{y \mid y$ is a successor of $x\}$.

In this paper, we consider a definition of P systems, which extends earlier versions of tissue and neural P systems [11, 15].

**Definition 1.** A P system of order $n$ with duplex channels and cell states is a system $\Pi = (O, K, \delta)$, where:

1. $O$ is a finite non-empty alphabet of objects;
2. $K = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is a finite set of cells;
3. $\delta$ is an irreflexive binary relation on $K$, which represents a set of structural arcs between cells, with duplex communication capabilities.

Each cell, $\sigma_i \in K$, has the initial configuration $\sigma_i = (Q_i, s_{i0}, w_{i0}, R_i)$, and the current configuration $\sigma_i = (Q_i, s_i, w_i, R_i)$, where:

- $Q_i$ is a finite set of states;
- $s_{i0} \in Q_i$ is the initial state; $s_i \in Q_i$ is the current state;
- $w_{i0} \in O^*$ is the initial content; $w_i \in O^*$ is the current content; note that, for $o \in O$, $|w_i|_o$ denotes the multiplicity of object $o$ in the multiset $w_i$;
- $R_i$ is a finite linearly ordered set of multiset rewriting rules (with promoters) of the form: $s x \rightarrow_\alpha s' x' (u)_\beta | z$, where $s, s' \in Q$, $x, x', u \in O^*$, $z \in O^*$, $\beta \in \{\min, \max\}$ is a rewriting operator and $\beta \in \{\uparrow, \downarrow, \leftrightarrow\}$ is a transfer operator (in this model, each rule has its own rewriting and transfer operators). For convenience, we also allow a rule to contain zero or more instances of $(u)_\beta$. If $u = z = \lambda$, i.e. the empty multiset of objects, this rule can be abbreviated as $s x \rightarrow_\alpha s' x'$.

A cell evolves by applying one or more rules, which can change its content and state and can send objects to its neighbors. For a cell $\sigma_i = (Q_i, s_i, w_i, R_i)$, a rule $s x \rightarrow_\alpha s' x' (u)_\beta | z \in R_i$ is applicable, if $s = s_i$, $x \subseteq w_i$, $z \subseteq w_i$, $\delta(i) \neq \emptyset$ for $\beta = \uparrow, \downarrow, \leftrightarrow$ and $\delta(i) \cup \delta^{-1}(i) \neq \emptyset$ for $\beta = \uparrow, \downarrow$.

The application of a rule transforms the current state $s$ to the target state $s'$, rewrites multiset $x$ as $x'$ and sends multiset $u$ as specified by the operator $\beta$ (as further described below). Note that, multisets $x'$ and $u$ will not be visible to other applicable rules in this same step, but they will be visible after all the applicable rules have been applied.

The rules are applied in the weak priority order [16], i.e. (1) higher priority applicable rules are assigned before lower priority applicable rules, and (2) a lower priority applicable rule is assigned only if it indicates the same target state as the previously assigned rules.

The operator $\alpha = \max$ indicates that an applicable rewriting rule of $R_i$ is applied as many times as possible. The operator $\alpha = \min$ indicates that an applicable rewriting rule of $R_i$ is applied once. If the right-hand side of a rule contains $(u)_\beta$, $\beta \in \{\uparrow, \downarrow, \leftrightarrow\}$, then for each application of this rule, a copy of
multiset $u$ is replicated and sent to each cell $\sigma_j \in \delta^{-1}(i)$ if $\beta = \uparrow$, $\sigma_j \in \delta(i)$ if $\beta = \downarrow$, and $\sigma_j \in \delta(i) \cup \delta^{-1}(i)$ if $\beta = \downarrow$. All applicable rules are synchronously applied in one step. An evolution of a P system is a sequence of steps, that starts from the initial configuration. An execution halts if no further rules are applicable for all cells.

**Problem 2.** We formulate the FSSP to P systems as follows:

**Input:** An integer $n \geq 2$ and an integer $g$, $1 \leq g \leq n$, indicating the general.

**Output:** A class $C$ of P systems that satisfies the following three conditions for any weakly connected digraph $(X, A)$, isomorphic to the structure of a member of $C$ with $n = |X|$ cells.

1. All cells have the same set of states $Q$ and set of rules $R$, and start from an initial quiescent state $s_q$. Additionally, except $\sigma_g$, all cells start with an empty initial content.
2. There exists state $s_f \in Q$, $s_f \neq s_q$, for all $\sigma_i \in K$, such that during the last step of the system’s evolution, all cells enter state $s_f$, simultaneously and for the first time.
3. Cell $\sigma_g$ is the only cell with an applicable rule (i.e. $\sigma_g$ can evolve) from its initial configuration.

We want to find a general-purpose solution to the FSSP that synchronizes in the fewest number of steps, as a function of some of the natural structural properties of a weakly-connected digraph $(X, A)$, such as the eccentricity of node $g \in X$ in the communication graph defined by $A$.

### 3 Deterministic FSSP solution for rooted trees

We first solve Problem 2 for the subclass of weakly-connected digraphs $(X, A)$, where the underlying graph of $(X, A)$ is a tree. This section is organized as follows. In Section 3.1, we present the P system for solving the FSSP for trees rooted at the general. In order to help the comprehension of our FSSP algorithm, we provide a trace of the FSSP algorithm in Table 1. Phase I of our FSSP algorithm is described in Section 3.2, which finds the middle cell (i.e. a center of a tree, closest to the root) and determines the height of the middle cell. Phase II of our FSSP algorithm is described in Section 3.3, which broadcasts the “command” that prompts all cells to enter the firing state. Finally, in Section 3.4, we present some empirical results that show improvements of our algorithm over the previously best-known FSSP algorithms for tree-based P systems [1, 5].

#### 3.1 P systems for solving the FSSP for rooted trees

Given a tree $(X, A)$ and $g \in X$, our FSSP algorithm is implemented using the P system $\Pi = (O, K, \delta)$ of order $n = |X|$, where:

1. $O = \{a, b, c, e, h, o, v, w\}$. 

2. $K = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$.

3. $\delta$ is a rooted tree, with an underlying graph isomorphic to $(X, A)$, where the general $\sigma_i \in K$ (the root of $\delta$) corresponds to $g \in X$.

All cells have the same set of states, the same set of rules and start at the same initial quiescent state $s_0$, but with different initial contents. The third output condition of Problem 2 will be satisfied by our chosen set of rules.

For each cell $\sigma_i \in K$, its initial configuration is $\sigma_i = (Q, s_0, w_0, R)$ and its final configuration at the end of the execution is $\sigma_i = (Q, s_6, \emptyset, R)$, where:

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, where $s_0$ is the initial quiescent state and $s_6$ is the firing state.
- $w_{i0} = \{\emptyset\}$ if $\sigma_i = \sigma_g$.
- $R$ is defined by the following rulesets.

**Rules used in Phase I:** all the rules in states $s_0, s_1, s_2, s_3$ and rule 4.6 in state $s_4$.

**Rules used in Phase II:** all the rules in states $s_4$ and $s_5$, except rule 4.6.

0. Rules in state $s_0$:
   1. $s_0 a \rightarrow_{\text{max}} s_1 ahou (b)_{\downarrow}$
   2. $s_0 b \rightarrow_{\text{max}} s_1 ahv (c)_{\uparrow} (b)_{\downarrow}$
   3. $s_0 b \rightarrow_{\text{max}} s_4 a (ceh)_{\uparrow}$

1. Rules in state $s_1$:
   1. $s_1 a \rightarrow_{\text{max}} s_2 ah$

2. Rules in state $s_2$:
   1. $s_2 aac \rightarrow_{\text{max}} s_4 a$
   2. $s_2 aa \rightarrow_{\text{max}} s_3 a$
   3. $s_2 cce \rightarrow_{\text{max}} s_2$
   4. $s_2 cc \rightarrow_{\text{max}} s_2$
   5. $s_2 ace \rightarrow_{\text{max}} s_3 aceh$
   6. $s_2 acwoo \rightarrow_{\text{max}} s_2 aa (o)_{\downarrow}$
   7. $s_2 acou \rightarrow_{\text{max}} s_2 aa (a)_{\downarrow}$
   8. $s_2 aco \rightarrow_{\text{max}} s_2 aehoo$
   9. $s_2 ao \rightarrow_{\text{max}} s_2 aaaa$
   10. $s_2 ae \rightarrow_{\text{max}} s_2 aeh$
   11. $s_2 a \rightarrow_{\text{max}} s_2 aa (c)_{\uparrow}$
   12. $s_2 u \rightarrow_{\text{max}} s_2$

3. Rules in state $s_3$:
   1. $s_3 a \rightarrow_{\text{max}} s_4 a$
   2. $s_3 h \rightarrow_{\text{max}} s_4$

4. Rules in state $s_4$:
   1. $s_4 hh \rightarrow_{\text{max}} s_5 w (v)_{\uparrow}$
   2. $s_4 avw \rightarrow_{\text{max}} s_5 aw (v)_{\uparrow}$
   3. $s_4 avv \rightarrow_{\text{max}} s_5 aw$
   4. $s_4 av \rightarrow_{\text{max}} s_5$
   5. $s_4 v \rightarrow_{\text{max}} s_5 w (v)_{\uparrow}$
   6. $s_4 o \rightarrow_{\text{max}} s_4$

5. Rules in state $s_5$:
   1. $s_5 aww \rightarrow_{\text{max}} s_5 aw$
   2. $s_5 aw \rightarrow_{\text{max}} s_6$
   3. $s_5 v \rightarrow_{\text{max}} s_6$
   4. $s_5 o \rightarrow_{\text{max}} s_6$
Table 1. The traces of the FSSP algorithm on a P system with the membrane structure defined by the tree shown in Figure 1...
Fig. 1. (a) a tree with the center $\sigma_5$; (b) a tree with two centers $\sigma_3$ and $\sigma_5$, $\sigma_3$ being the middle cell.

3.2 Phase I: Find the middle cell of rooted trees

In this phase, a breadth-first search (BFS) is performed from the root, which propagates symbol $b$ from the root to all other cells. When the symbol $b$ from the BFS reaches a leaf cell, symbol $c$ is reflected back up the tree. Starting from the root, the search for the middle cell is performed as described below, where symbol $o$ represents the current search pivot. Note that symbol $o$'s propagation speed is $1/3$ of the propagation speed of symbols $b$ and $c$; intuitively, this ensures that $o$ and $c$ meet in the middle cell.

We provide a visual description of the propagations of symbols $b$, $c$ and $o$ in Figure 2 (for a tree with one center) and Figure 3 (for a tree with two centers).

Details of Phase I

Objective: The objective of Phase I is to find the middle cell, $\sigma_m$, and its height, $\text{height}_g(m)$.

Precondition: Phase I starts with the initial configuration of P system $\Pi$, described in Section 3.1.

Postcondition: Phase I ends when $\sigma_m$ enters state $s_4$. At the end of Phase I, the configuration of cell $\sigma_i \in K$ is $(Q, s_4, w_i, R)$, where $|w_i|_a = 1$; $|w_i|_b = 2 \cdot \text{height}_g(i)$, if $\sigma_i = \sigma_m$.

Description: In Phase I, each cell starts in state $s_0$, transits through states $s_1, s_2, s_3$, and ends in state $s_4$; a cell in state $s_4$ will ignore any symbol $o$ that it may receive.

The behaviors of cells in this phase are described below.

- **Propagation of symbol $b$**: The root cell sends symbol $b$ to all its children (rule 0.1). An internal cell forwards the received symbol $b$ to all its children.
Fig. 2. Propagations of symbols $b$, $c$ and $o$, in a tree with one center. The symbols $c$ and $o$ meet at the middle cell $\sigma_5$. Cells that have sent symbol $c$ or $o$ are shaded. The propagation of symbol $o$ to a shaded cell is omitted. In cell $\sigma_j$, $j \in \{1,3\}$, $|w_j|_o - 1$ represents the number of steps since $\sigma_j$ received symbol $c$ from all of its children but one.
Fig. 3. Propagations of symbols $b$, $c$ and $o$, in a tree with two centers. The symbols $c$ and $o$ meet at the middle cell $\sigma_3$. Cells that have sent symbol $c$ or $o$ are shaded. The propagation of symbol $o$ to a shaded cell is omitted. In cell $\sigma_j$, $j \in \{1,3\}$, $|w_j|_o - 1$ represents the number of steps since $\sigma_j$ received symbol $c$ from all of its children but one.
(rule 0.2) After applying rule 0.1 or 0.2, each of these non-leaf cells produces a copy of symbol $h$ in each step, until it receives symbol $c$ from all its children (rules 1.1, 2.5 and 2.10).

- **Proposition 3.** Cell $\sigma_m$ receives the symbol $c$ from all its children by step $\text{height}_g(g) + \text{height}_g(m)$.

  **Proof.** Cell $\sigma_m$ is at distance $\text{height}_g(g) - \text{height}_g(m)$ from $\sigma_g$, hence $\sigma_m$ receives symbol $b$ in step $\text{height}_g(g) - \text{height}_g(m)$. In the subtree rooted at $\sigma_m$, the propagations of the symbol $b$ from $\sigma_m$ to its farthest leaf and the symbol $c$ reflected from the leaf to $\sigma_m$ take $2 \cdot \text{height}_g(m)$ steps. Thus, $\sigma_m$ receives symbol $c$ from all its children by step $\text{height}_g(g) - \text{height}_g(m) + 2 \cdot \text{height}_g(m) = \text{height}_g(g) + \text{height}_g(m)$.

- **Proposition 4.** The propagation of the symbol $a$ from $\sigma_g$ to $\sigma_m$ takes at most $\text{height}_g(g) + \text{height}_g(m)$ steps.

  **Proof.** For a given tree $T_g$, rooted at $\sigma_g$, we construct a tree $T_m$, which reroots $T_g$ at $\sigma_m$. Recall, $T_m(i)$ denotes a subtree rooted at $\sigma_i$ in $T_m$. Assume that $\sigma_m$ has $k \geq 2$ subtrees, $T_m(1), T_m(2), \ldots, T_m(k)$, such that $\text{height}_m(1) \geq$
Fig. 4. (a) k subtrees of $\sigma_m$, $T_m(1), T_m(2), \ldots, T_m(k)$. (b) The structure of subtree $T_m(j)$, which contains $\sigma_g$.

height$_m(2) \geq \cdots \geq$ height$_m(k)$ and height$_m(1) -$ height$_m(2) \leq 1$. Figure 4 (a) illustrates the subtrees of $\sigma_m$.

Assume $T_m(i)$ is a subtree of $\sigma_m$, which contains $\sigma_g$. In $T_m(i)$, let $z$ be the height of $\sigma_g$ and $x + w \geq 0$ be the distance between $\sigma_g$ and $\sigma_i$. Figure 4 (b) illustrates the $z$, $x$ and $w$ in $T_m(i)$.

To prove Proposition 4, we determine the number of steps needed to propagate symbol $o$ from $\sigma_g$ to $\sigma_m$. In $T_m(i)$, let $p$ be a path from $\sigma_i$ to its farthest leaf and $t$ be the number of steps needed to propagate symbol $o$ from $\sigma_g$ to $\sigma_m$. Note, height$_m(m) =$ height$_g(m)$ and $x + w + 1 =$ height$_g(g) -$ height$_g(m)$.

- If $\sigma_g$ is a part of path $p$, then $z+x+w+1 =$ height$_m(i)+1 =$ height$_m(m)-j$, $j \geq 0$, and $t = 2z + 3(x + w + 1)$. Hence,
  \[
t = 2(z + x + w + 1) + (x + w + 1) = 2(\text{height}_m(m) - j) + (\text{height}_g(g) - \text{height}_g(m)) = \text{height}_g(g) + \text{height}_g(m) - 2j
  \]

- If $\sigma_g$ is not a part of $p$, then $z+x+w+1 < v+w+1 =$ height$_m(i)+1 =$ height$_m(m)-j$, $j \geq 0$, and $t = x + 2v + 3(w + 1)$. Hence,
  \[
t = 2(v + w + 1) + (x + w + 1) = 2(\text{height}_m(m) - j) + (\text{height}_g(g) - \text{height}_g(m)) = \text{height}_g(g) + \text{height}_g(m) - 2j
  \]

\Proposition{5} Phase I takes $\text{height}_g(g) + \text{height}_g(m) + 2$ steps.

\Proof From Propositions 3 and 4, symbols $o$ and $c$ meets in $\sigma_m$ at step $\text{height}_g(g) + \text{height}_g(m)$. Cell $\sigma_m$ enters state $s_4$ by applying rule 2.9 and 2.1, which takes two steps. Thus, Phase I takes $\text{height}_g(g) + \text{height}_g(m) + 2$ steps. \qed
3.3 Phase II: Determine the step to enter the firing state

Phase II begins immediately after Phase I. In Phase II, the middle cell broadcasts the “firing” order, which prompts receiving cells to enter the firing state. In general, the middle cell does not have direct communication channels to all cells. Thus, the firing order has to be relayed through intermediate cells, which results in some cells receiving the order before other cells. To ensure that all cells enter the firing state simultaneously, each cell needs to determine the number of steps it needs to wait, until all other cells receive the order.

The firing order is paired with a **counter**, which is initially set to the eccentricity of the middle cell. Propagating an order from one cell to another decrements its current counter by one. The current counter of the received order equals the number of remaining steps before all other cells receive the order. Hence, each cell waits according to the current counter, before it enters the firing state. Figure 5 illustrates the propagation of the firing order.

---

**Fig. 5.** Propagations of the firing order from the middle cell, $\sigma_5$, where the counter is represented by the multiplicity of symbol $v$. Cells that have propagated the order are shaded. The propagation of symbol $v$ to a shaded cell is omitted. Let $k$ be the value of the counter that reaches $\sigma_i$ for the first time. Then $\sigma_i$ waits $k - 1$ steps, before it enters the firing state. The multiplicity of symbol $w$, which is decremented by one in each step, represents the number of remaining steps before a cell enters the firing state. In Step 5 (not shown here), all cells enter the firing state.

---

**Details of Phase II**

**Objective:** The objective of Phase II is to determine the step to enter the firing state, such that during the last step of Phase II, i.e. the system’s execution, all cells enter the firing state, simultaneously and for the first time.

**Precondition:** Phase II starts with the postcondition of Phase I, described in Section 3.2.
**Postcondition:** Phase II ends when all cells enter the firing state $s_6$. At the end of Phase II, the configuration of cell $c \in K$ is $(Q, s_6, \emptyset, R)$.

**Description:** The behaviors of the middle cell $c_m$ and a non-middle cell, $c_i \neq c_m$, in this phase are as follow. We also indicate which rules accomplish the described behaviors.

- We first describe the behavior of $c_m$. For every two copies of symbol $h$, $c_m$ produces one copy of symbol $w$ and sends one copy of symbol $v$ to all its neighbors (rule 4.1). In the next sequence of steps, $c_m$ consumes one copy of symbol $w$ (rule 5.1). If $c_m$ consumes all copies of symbol $w$, then $c_m$ enters the firing state (rule 5.2).
- Next, we describe the behavior of $c_i \neq c_m$. Let $k_i \geq 1$ denote the multiplicity of symbol $v$ that $c_i$ receives for the first time. If $k_i = 1$, then $c_i$ enters the firing state (rule 4.4). If $k_i \geq 2$, then $c_i$ consumes $k_i$ copies of symbol $v$, produces $k_i - 1$ copies of symbol $w$ and sends $k_i - 1$ copies of symbol $v$ to all its neighbors (rules 4.2, 4.3 and 4.5); in each subsequent step, $c_i$ consumes one copy of symbol $w$ (rule 5.1) and $c_i$ enters the firing state (rule 5.2), after all copies of symbol $w$ is consumed.

**Proposition 6.** Cell $c_m$ produces $\text{height}_g(m)$ copies of symbol $w$ and sends $\text{height}_g(m)$ copies of symbol $v$ to all is neighbors.

*Proof.* At the beginning of Phase II, $c_m$ contains $2 \cdot \text{height}_g(m)$ copies of symbol $h$. As described earlier, for every two copies of the symbol $h$ that $c_m$ consumes, $c_m$ produces one copy of symbol $w$ and sends one copy of symbol $v$ to all its neighbors. \qed

**Proposition 7.** Cell $c_i$ receives $k$ copies of symbol $v$ at step $t$ and sends $k - 1$ copies of symbol $v$ to all its neighbors at step $t + 1$, where $k = \text{height}_g(m) - \text{depth}_m(i) + 1$ and $t = \text{height}_g(g) + \text{height}_g(m) + \text{depth}_m(i) + 2$.

*Proof.* Proof by induction on $\text{depth}_m(i) \geq 1$. First, $c_m$ sends $\text{height}_g(m)$ copies of symbol $w$ to all its neighbors. Thus, each cell $c_i$, at distance 1 from $c_m$, receives $\text{height}_g(m)$ copies of symbol $v$. By rules 4.3, 4.4, 4.7 and 4.8, $c_i$ consumes $\text{height}_g(m)$ copies of symbol $v$, produces $\text{height}_g(m) - 1$ copies of symbol $w$ and sends $\text{height}_g(m) - 1$ copies of symbol $v$ to all its neighbors.

Assume that the induction hypothesis holds for each cell $c_j$ at distance $\text{depth}_m(j)$. Consider cell $c_{i_1}$, where $\text{depth}_m(i) = \text{depth}_m(j) + 1$. By the induction hypothesis, cell $c_j \in \text{Neighbor}(i)$, sends $\text{height}_g(m) - \text{depth}_m(j) = \text{height}_g(m) - \text{depth}_m(i) + 1$ copies of symbol $v$, such that $c_j$ receives $\text{height}_g(m) - \text{depth}_m(i) + 1$ copies of symbol $v$. By rules 4.3, 4.4, 4.7 and 4.8, $c_i$ consumes $\text{height}_g(m) - \text{depth}_m(i) + 1$ copies of symbol $v$, produces $\text{height}_g(m) - \text{depth}_m(i)$ copies of symbol $w$ and sends $\text{height}_g(m) - \text{depth}_m(i)$ copies of symbol $v$ to all its neighbors. \qed
Proposition 8. Phase II takes $\text{height}_g(m) + 1$ steps.

Proof. Each cell $\sigma_i$ receives $\text{height}_g(m) - \text{depth}_m(i) + 1$ copies of symbol $v$ at step $\text{height}_g(g) + \text{height}_g(m) + \text{depth}_m(i) + 2$.

Consider $\sigma_j$, where $\text{depth}_m(j) = \text{height}_g(m)$. Cell $\sigma_j$ receives one copy of symbol $v$. As described earlier, if a cell receives one copy of symbol $v$, then it enters the firing state at the next step. Hence, $\sigma_j$ enters the firing state at step $\text{height}_g(g) + 2 \cdot \text{height}_g(m) + 3$.

Consider $\sigma_k$, where $\text{depth}_m(k) < \text{height}_g(m)$. Cell $\sigma_k$ contains $\text{height}_g(m) - \text{depth}_m(i)$ copies of symbol $w$ at step $\text{height}_g(g) + \text{height}_g(m) + \text{depth}_m(i) + 3$. Since $\sigma_k$ consumes one copy of symbol $w$ in each step, $\sigma_k$ will take $\text{height}_g(m) - \text{depth}_m(i)$ steps to consume all copies of symbol $w$. Hence, $\sigma_j$ enters the firing state at step $(\text{height}_g(g) + \text{height}_g(m) + \text{depth}_m(i) + 3) + (\text{height}_g(m) - \text{depth}_m(i)) = \text{height}_g(g) + 2 \cdot \text{height}_g(m) + 3$.

Phase I ends at step $\text{height}_g(g) + \text{height}_g(m) + 2$ and all cells enter the firing state at step $\text{height}_g(g) + 2 \cdot \text{height}_g(m) + 3$. Thus, Phase II takes $\text{height}_g(m) + 1$ steps.

\[ \square \]

Theorem 9. The synchronization time of our FSSP solution, for a P system with underlying structure of a tree, is $\text{height}_g(g) + 2 \cdot \text{height}_g(m) + 3$.

Proof. The result is obtained by summing the individual running times of Phases I and II, as given by Propositions 5 and 8: $(\text{height}_g(g) + \text{height}_g(m) + 2) + (\text{height}_g(m) + 1) = \text{height}_g(g) + 2 \cdot \text{height}_g(m) + 3$. \[ \square \]

3.4 Empirical results

We tested the improvement in running times over the previously best-known FSSP algorithms that synchronize tree-based P systems [1, 5]. We wanted to see how our new running time, that is proportional to $e + 2r$, compares with the earlier value of $3e$, where $e$ is the eccentricity of the general (which is also the height of the tree, rooted at the general) and $r$ is the radius of a tree. We did two tests suites; one for relatively small trees and one for larger trees as shown in Tables 2 and 3, respectively. In both cases, our empirical results show at least 20% reduction in the number of steps needed to synchronize, which we believe is significant.

For the statistics given in Table 2, we generated random (free) trees by starting from a single node and repeatedly add new leaf nodes to the partially generated tree. We then averaged over all possible locations for the general node. The “average gain” is the average difference $3e - (e + 2r)$ and the “average % gain” is improvement as a percentage speedup over 3e.

For the statistics given in Table 3, we generated random labeled trees using the well-known Pr"ufer correspondence [21] (using the implementation given in Sage [18]). In these sets of trees, the first indexed vertex is randomly placed, unlike the random trees generated in our first test suite. Hence, for this test
Table 2. Statistics for improvement on many random trees of various (smaller) orders.

<table>
<thead>
<tr>
<th>n</th>
<th>average height</th>
<th>average radius</th>
<th>average 3-height</th>
<th>average height+2-radius</th>
<th>average % gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>22.12</td>
<td>14.49</td>
<td>66.36</td>
<td>51.1</td>
<td>23.00</td>
</tr>
<tr>
<td>200</td>
<td>31.91</td>
<td>21.35</td>
<td>95.73</td>
<td>74.61</td>
<td>22.06</td>
</tr>
<tr>
<td>300</td>
<td>41.13</td>
<td>26.79</td>
<td>123.39</td>
<td>94.71</td>
<td>23.24</td>
</tr>
<tr>
<td>400</td>
<td>47.86</td>
<td>31.3</td>
<td>143.58</td>
<td>110.46</td>
<td>23.07</td>
</tr>
<tr>
<td>500</td>
<td>51.52</td>
<td>33.77</td>
<td>154.56</td>
<td>119.06</td>
<td>22.97</td>
</tr>
<tr>
<td>600</td>
<td>57.16</td>
<td>37.76</td>
<td>171.48</td>
<td>132.68</td>
<td>22.63</td>
</tr>
<tr>
<td>700</td>
<td>63.43</td>
<td>42.19</td>
<td>190.29</td>
<td>147.81</td>
<td>22.32</td>
</tr>
<tr>
<td>800</td>
<td>68.12</td>
<td>45.37</td>
<td>204.36</td>
<td>158.86</td>
<td>22.26</td>
</tr>
<tr>
<td>900</td>
<td>72.46</td>
<td>47.83</td>
<td>217.38</td>
<td>168.12</td>
<td>22.66</td>
</tr>
<tr>
<td>1000</td>
<td>79.94</td>
<td>52.21</td>
<td>239.82</td>
<td>184.36</td>
<td>23.13</td>
</tr>
</tbody>
</table>

Table 3. Statistics for improvement on random trees of various (larger) orders.

<table>
<thead>
<tr>
<th>n</th>
<th>diameter</th>
<th>radius</th>
<th>average eccentricity</th>
<th>average gain</th>
<th>average % gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>21</td>
<td>11</td>
<td>16.27</td>
<td>10.54</td>
<td>21.59</td>
</tr>
<tr>
<td>2000</td>
<td>32</td>
<td>16</td>
<td>23.45</td>
<td>14.90</td>
<td>21.18</td>
</tr>
<tr>
<td>3000</td>
<td>26</td>
<td>13</td>
<td>19.97</td>
<td>13.95</td>
<td>23.27</td>
</tr>
<tr>
<td>4000</td>
<td>30</td>
<td>15</td>
<td>22.65</td>
<td>15.30</td>
<td>22.51</td>
</tr>
<tr>
<td>5000</td>
<td>35</td>
<td>18</td>
<td>26.51</td>
<td>17.01</td>
<td>21.40</td>
</tr>
<tr>
<td>6000</td>
<td>32</td>
<td>16</td>
<td>23.82</td>
<td>15.64</td>
<td>21.89</td>
</tr>
<tr>
<td>7000</td>
<td>34</td>
<td>17</td>
<td>25.29</td>
<td>16.58</td>
<td>21.85</td>
</tr>
<tr>
<td>8000</td>
<td>34</td>
<td>17</td>
<td>25.01</td>
<td>16.03</td>
<td>21.36</td>
</tr>
<tr>
<td>9000</td>
<td>40</td>
<td>20</td>
<td>28.37</td>
<td>16.74</td>
<td>19.67</td>
</tr>
<tr>
<td>10000</td>
<td>40</td>
<td>19</td>
<td>27.16</td>
<td>16.32</td>
<td>20.03</td>
</tr>
<tr>
<td>10000</td>
<td>38</td>
<td>19</td>
<td>27.36</td>
<td>16.72</td>
<td>20.37</td>
</tr>
<tr>
<td>20000</td>
<td>37</td>
<td>19</td>
<td>28.23</td>
<td>18.47</td>
<td>21.80</td>
</tr>
<tr>
<td>30000</td>
<td>43</td>
<td>22</td>
<td>31.74</td>
<td>19.49</td>
<td>20.46</td>
</tr>
<tr>
<td>40000</td>
<td>43</td>
<td>22</td>
<td>31.55</td>
<td>19.09</td>
<td>20.18</td>
</tr>
<tr>
<td>50000</td>
<td>42</td>
<td>21</td>
<td>30.81</td>
<td>19.63</td>
<td>21.23</td>
</tr>
<tr>
<td>60000</td>
<td>44</td>
<td>22</td>
<td>32.50</td>
<td>21.00</td>
<td>21.54</td>
</tr>
<tr>
<td>70000</td>
<td>48</td>
<td>24</td>
<td>34.55</td>
<td>21.09</td>
<td>20.35</td>
</tr>
<tr>
<td>80000</td>
<td>45</td>
<td>23</td>
<td>33.08</td>
<td>20.17</td>
<td>20.32</td>
</tr>
<tr>
<td>90000</td>
<td>50</td>
<td>25</td>
<td>36.00</td>
<td>22.01</td>
<td>20.37</td>
</tr>
<tr>
<td>100000</td>
<td>47</td>
<td>24</td>
<td>34.15</td>
<td>20.29</td>
<td>19.81</td>
</tr>
</tbody>
</table>
suite, we did not need to average over all possible general node locations per tree. Due to the uniform randomness of the labeled tree generator, we assumed the general is placed at the node labeled by 1. Each row in Table 3 is based on 100 random trees of that given order.

We have run both test suites several times and the results are consistent with these two tables. Hence, we are pretty confident in the practical speedup that our new synchronization algorithm provides.

4 FSSP solution for digraphs

The key idea of FSSP solution for digraphs is as follows. For a given digraph, perform a BFS from the general on the communication graph and construct a virtual spanning tree, implemented via pointer symbols, not by changing existing arcs. If a node finds multiple parents in the BFS, then one of the parents is chosen as its spanning tree parent. In Figure 6, (a) illustrates a digraph $G$, (b) illustrates the underlying graph of $G$ and (c) illustrates a spanning tree of the underlying graph of $G$, rooted at $\sigma_1$.

Using the spanning tree constructed from the BFS, the FSSP algorithm described in Section 3, is applied to achieve the synchronization.

We present the details of P system for solving the FSSP (Problem 2) for digraphs in Section 4.1. A trace of the FSSP algorithm for digraphs is given in Table 4. The details Phases I and II of this FSSP algorithm are described in Sections 4.2 and 4.3, respectively. Finally, in Section 4.4, we present some empirical results that illustrates expected improvements of our new algorithm over our previous FSSP algorithm for digraphs [5].

![Fig. 6.](image_url)

Fig. 6. (a) A digraph $G$. (b) The underlying graph of $G$. (c) A spanning tree of the underlying graph of $G$, rooted at $\sigma_1$. 
4.1 P systems for solving the FSSP for digraphs

Given a digraph \((X,A)\) and \(g \in X\), our FSSP algorithm is implemented using the P system \(\Pi' = (O,K,\delta)\) of order \(n = |X|\), where:

1. \(O = \{a,h,o,v,w,x,z\} \cup \{\iota_k | 1 \leq k \leq n\}\).
2. \(K = \{\sigma_1,\sigma_2,\ldots,\sigma_n\}\).
3. \(\delta\) is a digraph, isomorphic to \((X,A)\), where the general \(\sigma_g \in K\) corresponds to \(g \in X\).

All cells have the same set of states and start at the same initial quiescent state \(s_0\), but with different initial contents and set of rules. The third output condition of Problem 2 will be satisfied by our chosen set of rules.

In this FSSP solution, we extend the basic P system framework, described Section 2. Specifically, we assume that each cell \(\sigma_i \in K\) has a unique cell ID symbol \(\iota_i\), which will be used as an immutable promoter and we allow rules with a simple form of complex symbols.

To explain these additional features, consider rules 3.10 and 3.11 from the ruleset \(R\), listed below. In this ruleset, symbols \(i\) and \(j\) are free variables (which in our case happen to match cell IDs). Symbols \(e_i\) and \(e_j\) are complex symbols. Rule 3.11 deletes all existing \(e_j\) symbols, regardless of the actual values matched by the free variable \(j\). However, the preceding rule 3.10 fires only for symbols \(e_i\), with indices \(i\) matching the local cell ID, as required by the right-hand side promoter \(\iota_i\). Together, rules 3.10 and 3.11, applied in a weak priority scheme, keep all symbols \(e_i\), with indices \(i\) matching the local cell ID, and delete all other symbols \(e_j\).

Remark 10. Cell IDs seem to reduce the general-applicability of our solution, since alphabet \(O\) seems to depends on \(|X|\). However, in future, we foresee that P systems will allow multisets of structured objects, such as tuples. Hence, we can encode cell IDs as binary strings over alphabet of size two, which is independent of \(|X|\).

For each cell \(\sigma_i \in K\), its initial configuration is \(\sigma_i = (Q,s_0,w_{i0},R)\) and its final configuration at the end of the execution is \(\sigma_i = (Q,s_7,\{\iota_i\},R)\), where:

- \(Q = \{s_0,s_1,s_2,s_3,s_4,s_5,s_6,s_7\}\), where \(s_0\) is the initial quiescent state and \(s_7\) is the firing state.
- \(w_{i0} = \{\iota_g\} \) if \(\sigma_i = \sigma_g\),
- \(\{\iota_i\} \) if \(\sigma_i \neq \sigma_g\).
- \(R\) is defined by the following rulesets.

Rules used in Phase I: all the rules in states \(s_0, s_1, s_2, s_3, s_4\) and rules 5.5 and 5.6 in state \(s_5\).

Rules used in Phase II: all the rules in states \(s_5\) and \(s_6\), except rules 5.5 and 5.6.
0. Rules for cells in state $s_0$:
1. $s_0 \rightarrow_{\text{bin}} s_1 a o (x b_i)_t \| i_i$
2. $s_0 x \rightarrow_{\text{bin}} s_1 a (x b_i)_t \| i_i$
3. $s_0 b_j \rightarrow_{\text{max}} s_1 p_j$

1. Rules for cells in state $s_1$:
1. $s_1 a p_j \rightarrow_{\text{max}} s_2 a p_j (e_j)_t$
2. $s_1 a \rightarrow_{\text{max}} s_2 a$
3. $s_1 p_j \rightarrow_{\text{max}} s_2$

2. Rules for cells in state $s_2$:
1. $s_2 a \rightarrow_{\text{max}} s_3 a$
2. $s_2 b_j \rightarrow_{\text{max}} s_3$
3. $s_2 x \rightarrow_{\text{max}} s_3$

3. Rules for cells in state $s_3$:
1. $s_3 a a a \rightarrow_{\text{max}} s_5 a$
2. $s_3 a a \rightarrow_{\text{max}} s_4 a$
3. $s_3 c_i e_i \rightarrow_{\text{max}} s_3 |i_i$
4. $s_3 a o e_i \rightarrow_{\text{max}} s_3 a a (o)_t |i_i$
5. $s_3 a e_i e_i \rightarrow_{\text{max}} s_3 a h o e_i e_i |i_i$
6. $s_3 a o e_i \rightarrow_{\text{max}} s_3 a h o o e_i |i_i$
7. $s_3 a o \rightarrow_{\text{max}} s_3 a a a$
8. $s_3 a e_i \rightarrow_{\text{max}} s_3 a e_i h |i_i$
9. $s_3 a p_j \rightarrow_{\text{max}} s_3 a a (c_j)_t$
10. $s_3 e_i \rightarrow_{\text{max}} s_3 e_i |i_i$
11. $s_3 c_j \rightarrow_{\text{max}} s_3$
12. $s_3 p_j \rightarrow_{\text{max}} s_4$
13. $s_3 p_j \rightarrow_{\text{max}} s_5$

4. Rules for cells in state $s_4$:
1. $s_4 a \rightarrow_{\text{max}} s_5$
2. $s_4 h \rightarrow_{\text{max}} s_5$
3. $s_4 c_j \rightarrow_{\text{max}} s_5$

5. Rules for cells in state $s_5$:
1. $s_5 a \rightarrow_{\text{max}} s_6 a (z)_t$
2. $s_5 h h \rightarrow_{\text{max}} s_6 w (v)_t$
3. $s_5 z v \rightarrow_{\text{max}} s_6 a (z)_t$
4. $s_5 v \rightarrow_{\text{max}} s_6 w (v)_t$
5. $s_5 o \rightarrow_{\text{max}} s_5$
6. $s_5 c_j \rightarrow_{\text{max}} s_5$

6. Rules for cells in state $s_6$:
1. $s_6 a w \rightarrow_{\text{max}} s_6 a$
2. $s_6 a \rightarrow_{\text{max}} s_7$
3. $s_6 a \rightarrow_{\text{max}} s_7$
4. $s_6 a \rightarrow_{\text{max}} s_7$

4.2 Phase I: Find the middle cell of a BFS spanning tree

For a given digraph-based P system, a (virtual) spanning tree is constructed by a standard BFS originated from the general, where the tree parent of each cell is one of its BFS parents (randomly chosen). Each cell keeps the track of its spanning tree parent and this is achieved by the use of cell IDs (unique identifier ID), e.g., $i$ is the cell ID of $\sigma_i$.

Details of Phase I

Objective: The objective of Phase I is to find the middle cell, $\sigma_m$, and its height, $\text{height}_p(m)$.

Precondition: Phase I starts with the initial configuration of P system $II$, described in Section 4.1.

Postcondition: Phase I ends when $\sigma_m$ enters state $s_5$. At the end of Phase I, the configuration of cell $\sigma_i \in K$ is $(Q, s_5, w_i, R)$, where $|w_i|_i = 1; |w_i|_a = 1$ and $|w_i|_h = 2 \cdot \text{height}_p(i)$, if $\sigma_i = \sigma_m$. 
Table 4. The traces of the FSSP algorithm on the digraph of Figure 6 (a), where the general is $\sigma_1$ and the middle cell is $\sigma_2$. The step in which the Phase I ends (or the Phase II begins) is indicated by the shaded table cells.

<table>
<thead>
<tr>
<th>Step</th>
<th>$\sigma_1$</th>
<th>$\sigma_2$</th>
<th>$\sigma_3$</th>
<th>$\sigma_4$</th>
<th>$\sigma_5$</th>
<th>$\sigma_6$</th>
<th>$\sigma_7$</th>
<th>$\sigma_8$</th>
<th>$\sigma_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>1</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>2</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>3</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>4</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>5</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>6</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>7</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>8</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>9</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>10</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>11</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>12</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>13</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>14</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>15</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>16</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
<tr>
<td>17</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>$s_8$</td>
<td>$s_9$</td>
</tr>
</tbody>
</table>
Description: We describe below the details of the BFS spanning tree construction and the propagation of the reflected symbol in the BFS tree. The symbol \( o \), starting from the general, propagates from a tree parent to one of its children, as described in the FSSP solution for tree-based P systems (Section 3.2). Hence, the details of symbol \( o \) propagation are not given here.

- **The details of the BFS spanning tree construction:**
  A BFS starts from the general. When the search reaches cell \( \sigma_i \), \( \sigma_i \) will send a copy of symbol \( b_i \) to all its neighbors (rule 0.1 or 0.2). From the BFS, cell \( \sigma_i \) receives a copy of symbol \( b_j \) from each \( \sigma_j \in \text{Pred}_g(i) \), where \( \sigma_j \) is a BFS dag parent of \( \sigma_i \). Cell \( \sigma_i \) temporarily stores all of its BFS dag parents by transforming each received symbol \( b_j \) to symbol \( p_j \) (rule 0.3). Note, \( \sigma_i \) will also receive a copy of symbol \( b_k \) from each \( \sigma_k \in \text{Peer}_g(i) \cup \text{Succ}_g(i) \); however, \( \sigma_i \) will discard each received symbol \( b_k \).
  Each cell selects one of its BFS dag parents as its tree parent. If cell \( \sigma_i \) has chosen \( \sigma_j \) as its tree parent, then \( \sigma_i \) will discard each \( p_k \), where \( \sigma_k \in \text{Pred}_g(i) \setminus \{ \sigma_j \} \) (rule 1.3). Additionally, \( \sigma_i \) will send a copy of symbol \( e_j \) to all its neighbors, which will be discarded by all \( \sigma_i \)'s neighbors, except \( \sigma_j \) (rule 1.1).
  Hence, in each cell \( \sigma_i \), the multiplicity of symbol \( e_i \) will indicate the number of \( \sigma_i \)'s tree children and symbol \( p_j \) will indicate that \( \sigma_j \) is the tree parent of \( \sigma_i \); also, symbol \( p_j \) will later be used to propagate the reflected symbol back up the tree.

- **The details of reflected symbol propagation:**
  To replicate the propagation of a reflected symbol up the BFS tree, each internal cell of the BFS tree needs to check if the received a reflected symbol came from one of its BFS tree children.
  Let \( \sigma_i \) be a BFS tree child of \( \sigma_j \), where \( |w_i|_{e_i} = 0 \). Recall that, in such case, cell \( \sigma_i \) contains symbol \( p_j \), where the subscript \( j \) is the ID of its BFS tree parent, and \( \sigma_j \) contains symbol \( e_j \), such that \( |w_j|_{e_j} \) is the number of \( \sigma_j \)'s BFS tree children.
  Guided by symbol \( p_j \), \( \sigma_i \) sends symbol \( e_j \) to all its neighbors (rule 3.9). Cell \( \sigma_j \) consumes a copy of symbol \( e_j \) with a copy of symbol \( e_j \) by rule 3.3; \( \sigma_j \) cannot consume symbol \( e_j \) with symbol \( e_k \), where \( j \neq k \). If \( \sigma_j \) receives symbol \( e_j \) from all its BFS tree children, then all copies of symbol \( e_j \) will be consumed, i.e. \( |w_j|_{e_j} = 0 \).
  Proposition 11 indicates the step in which the BFS reaches cell \( \sigma_i \) and \( \sigma_i \) receives symbol \( b_j \) from each \( \sigma_j \in \text{Pred}_g(i) \). Proposition 12 indicates the step in which \( \sigma_i \) receives symbol \( e_i \) from its tree child.

**Proposition 11.** Cell \( \sigma_i \) receives symbol \( b_j \) from each \( \sigma_j \in \text{Pred}_g(i) \) at step \( \text{depth}_g(i) \) and sends symbol \( b_j \) to all its neighbors at step \( \text{depth}_g(i) + 1 \).

**Proof.** Proof by induction, on \( d = \text{depth}_g(i) \geq 1 \). At step 1, the general \( \sigma_g \) sends symbol \( b_g \) to all its neighbors by rule 0.1. Hence, at step 1, each cell \( \sigma_k \) at depth 1 receives symbol \( b_g \). Then, at step 2, by rule 0.2, \( \sigma_k \) sends symbol \( b_k \) to each of its neighbors.
Assume that the induction hypothesis holds for each cell $\sigma_j$ at depth $d$. Consider cell $\sigma_i$ at depth $g(i) = m + 1 = \text{depth}_g(i) + 1$. By induction hypothesis, at step $\text{depth}_g(j) + 1$, each $\sigma_j \in \text{Pred}_g(i)$ sends symbol $b_j$ to all its neighbors. Thus, at step $\text{depth}_g(j) + 1 = \text{depth}_g(i)$, $\sigma_i$ receives symbol $b_j$. At step $\text{depth}_g(i) + 1$, by rule 0.2, $\sigma_i$ sends symbol $b_i$ to all its neighbors. 

**Proposition 12.** Cell $\sigma_i$ receives a copy of symbol $e_i$ from each of its tree children at step $\text{depth}_g(i) + 3$.

**Proof.** Assume that cell $\sigma_j \in \text{Succ}_g(i)$ has chosen $\sigma_i$ as its tree parent. From Proposition 11, cell $\sigma_j$ receives symbol $b_i$ at step $\text{depth}_g(j) = \text{depth}_g(i) + 1$. According to the description, $\sigma_j$ will send symbol $e_i$ at step $\text{depth}_g(j) + 2$. Thus, $\sigma_i$ will receive symbol $e_i$ at step $\text{depth}_g(i) + 3$. 

**Remark 13.** From Proposition 12, $\sigma_i$ receives symbol $e_i$ from its tree child at step $\text{depth}_g(i) + 3$. If $\sigma_i$ does not receive symbol $e_i$ at step $\text{depth}_g(i) + 3$, then $\sigma_i$ can recognize itself as a tree leaf and send a reflected symbol to its tree parent at step $\text{depth}_g(i) + 4$. That is, once a leaf cell is reached by the BFS, it will take three additional steps to send reflected symbol to its tree parent. Recall, in the FSSP algorithm for tree-based P systems, a leaf cell sends reflected symbol to its parent, one step after reached by the BFS. Thus, this FSSP algorithm for digraph-based P systems takes three additional steps to send the reflected symbol than the FSSP algorithm for tree-based P systems.

### 4.3 Phase II: Determine the step to enter the firing state

Similar to the Phase II described in Section 3.3, the firing order is broadcasted from the middle cell $\sigma_m$. The order is paired with a counter, which is initially set to the eccentricity of $\sigma_m$ and decrements by one in each step of this broadcast operation.

**Details of Phase II**

**Objective:** The objective of Phase II is to determine the step to enter the firing state, such that during the last step of Phase II, i.e. the system’s execution, all cells enter the firing state, simultaneously and for the first time.

**Precondition:** Phase II starts with the postcondition of Phase I, described in Section 4.2.

**Postcondition:** Phase II ends when all cells enter the firing state $s_T$. At the end of Phase II, the configuration of cell $\sigma_i \in K$ is $(Q, s_T, \{e_i\}, R)$.

**Description:** The order arrives in $\sigma_i$, along every shortest paths from $\sigma_m$ to $\sigma_i$. Hence, to compute the correct step to enter the firing state, cell $\sigma_i$ decrements, in each step, the sum of all received counter by the number of shortest paths from $\sigma_m$ to $\sigma_i$. $\sigma_i$ enters the firing state if the sum of all received counter becomes 0. The number of shortest paths from $\sigma_m$ to $\sigma_i$ is determined as follows.
Cell $\sigma_m$ sends a copy of symbol $z$. Each cell $\sigma_i$ forwards symbol $z$, received from each $\sigma_j \in \text{Pred}_m(i)$. The number of shortest paths from $\sigma_m$ to $\sigma_i$ is the sum of all copies of symbol $z$ that $\sigma_i$ receives from each $\sigma_j \in \text{Pred}_m(i)$.

Let $t$ be the current counter and $k$ be the number of shortest paths from $\sigma_m$ to the current cell. In the FSSP solution for tree-based P systems, the condition for entering the firing state in the next step is when $t = 1$ (note $k = 1$). However, the FSSP solution, as implemented in this section, cannot directly detect if $t = k$, since $k \geq 1$. Instead, a cell enters the firing state after $t = 0$ is detected. Thus, the FSSP algorithm for digraph-based P systems requires one additional step in Phase II.

**Theorem 14.** The synchronization time of the FSSP solution for digraph-based P systems is $\text{ecc}(g) + 2 \cdot \text{ecc}(m) + 7$.

**Proof.** This FSSP algorithm for digraph-based P systems requires four additional overhead steps than the FSSP algorithm for tree-based P systems. Three of these four overhead steps are described in Remark 13 and the remaining overhead step is mentioned in Section 4.3. \qed

We end this section with a comment regarding improving the communication requirements of our FSSP solution. Currently, there may be an exponential number of broadcast objects generated since a given cell currently receives a copy of the counter from every possible shortest path from the middle cell. We can reduce number of broadcasted counters from an exponential to a polynomial as follows. Assume that, a counter, sent or forwarded from a cell, is annotated with the cell’s ID. In Phase II, if a cell receives counter from its BFS tree neighbor (from a BFS tree child for cells on the path from the general to the middle cell, otherwise from its original BFS tree parent), then it broadcasts the reduced-by-one counter, now annotated with its own ID, to all its neighbors. The total number of steps of this revised algorithm would still be the same as given in Theorem 14.

4.4 Empirical results

We also tested the improvement in running times over our previous FSSP algorithm on digraph-based P systems. The rate of improvement drops off as the number of edges increase over $n-1$, the size of trees of order $n$. But for several sparse digraph structured P systems the improvement is still worthwhile.

We did two tests suites; one for relatively small digraphs (illustrated in Figure 7) and one for larger digraphs as shown in Table 5. The graphs used in our empirical tests were generated using NetworkX [8].

For the statistics given in Table 5, we first generated connected random graphs of order $n$ and size $m$. We then averaged over all possible locations for the general node. To model the parallel nature of P systems, we needed to generate a random BFS tree originating at the general. This was created by first performing a BFS from the general to constructing the BFS dag then randomly...
picking (for each non-general node) one parent within the dag structure as the parent for the BFS tree.

For this BFS tree, with \( e \) denoting the eccentricity of the general and \( r \) denoting the radius of the BFS tree, the “average gain” is the average difference of \( 3e - (e + 2r) \) and the “average % gain” is the average of the \( (3e - (e + 2r))/(3e) \) values. From our empirical results, we can observe that the radius of the BFS spanning trees seems to be close to the actual radius of the given virtual communication graphs.

For the statistics given in the three dimensional plots of Figure 7 (generated using Gnuplot [22]), we generated 100 random connected \((n,m)\)-graphs, for each order \(n\), \(20 \leq n \leq 40\), and size \(m = (n - 1) + 2k\), where \(0 \leq k \leq 20\). Note, the integer value of \(2k\) represents the number of edges added to a tree. We then averaged over all possible general starting positions. The vertical axis is the average percentage speedup of our new algorithm over our previous synchronization algorithm. One can also observe from this plot, at least 20% improvements (i.e. reduction in number of steps needed to synchronize), is maintained for \(k = 0\) (i.e. the graph is a tree). However, as the graphs become less sparse, the expected improvement drops to near zero, when as few as 40 edges are added to the trees. In general, for fixed \(k\), the expected improvement in performance, for \((n,n+k)\) digraphs slightly increases as \(n\) increases. However, for fixed \(n\), the expected improvement in performance drops drastically as \(k\) increases.

---

**Fig. 7.** Discrete 3-dimensional plot of expected synchronization improvements for a small range of random connected \((n,m)\)-graph structures, with \(m = (n - 1) + k\) edges.
Table 5. Statistics for reduction in number of steps needed to synchronize on a few random \((n,m)\)-graphs.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(m)</th>
<th>radius</th>
<th>gain</th>
<th>gain %</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>15</td>
<td>16.7</td>
<td>23.16</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>9</td>
<td>11.47</td>
<td>8.02</td>
</tr>
<tr>
<td>100</td>
<td>120</td>
<td>7</td>
<td>8.97</td>
<td>5.45</td>
</tr>
<tr>
<td>100</td>
<td>130</td>
<td>7</td>
<td>8.13</td>
<td>3.86</td>
</tr>
<tr>
<td>100</td>
<td>140</td>
<td>6</td>
<td>7.33</td>
<td>3.12</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>15</td>
<td>15.68</td>
<td>23.16</td>
</tr>
<tr>
<td>200</td>
<td>210</td>
<td>9</td>
<td>11.47</td>
<td>3.14</td>
</tr>
<tr>
<td>200</td>
<td>220</td>
<td>7</td>
<td>8.97</td>
<td>2.88</td>
</tr>
<tr>
<td>200</td>
<td>230</td>
<td>6</td>
<td>7.33</td>
<td>2.12</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
<td>20</td>
<td>20.73</td>
<td>17.91</td>
</tr>
<tr>
<td>300</td>
<td>310</td>
<td>16</td>
<td>19.12</td>
<td>5.08</td>
</tr>
<tr>
<td>300</td>
<td>320</td>
<td>13</td>
<td>15.74</td>
<td>3.9</td>
</tr>
<tr>
<td>300</td>
<td>330</td>
<td>12</td>
<td>15.0</td>
<td>3.37</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>24</td>
<td>24.56</td>
<td>21.94</td>
</tr>
<tr>
<td>400</td>
<td>410</td>
<td>22</td>
<td>24.79</td>
<td>7.73</td>
</tr>
<tr>
<td>400</td>
<td>420</td>
<td>19</td>
<td>21.91</td>
<td>7.12</td>
</tr>
<tr>
<td>400</td>
<td>430</td>
<td>15</td>
<td>17.85</td>
<td>5.37</td>
</tr>
<tr>
<td>400</td>
<td>440</td>
<td>13</td>
<td>15.86</td>
<td>4.48</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>28</td>
<td>29.14</td>
<td>23.30</td>
</tr>
<tr>
<td>500</td>
<td>510</td>
<td>24</td>
<td>27.28</td>
<td>9.68</td>
</tr>
<tr>
<td>500</td>
<td>520</td>
<td>19</td>
<td>23.17</td>
<td>3.58</td>
</tr>
<tr>
<td>500</td>
<td>530</td>
<td>16</td>
<td>19.87</td>
<td>5.68</td>
</tr>
<tr>
<td>500</td>
<td>540</td>
<td>17</td>
<td>19.25</td>
<td>5.70</td>
</tr>
<tr>
<td>600</td>
<td>600</td>
<td>28</td>
<td>30.99</td>
<td>22.35</td>
</tr>
<tr>
<td>600</td>
<td>610</td>
<td>25</td>
<td>28.78</td>
<td>14.63</td>
</tr>
<tr>
<td>600</td>
<td>620</td>
<td>22</td>
<td>24.96</td>
<td>5.39</td>
</tr>
<tr>
<td>600</td>
<td>630</td>
<td>19</td>
<td>22.06</td>
<td>5.72</td>
</tr>
<tr>
<td>600</td>
<td>640</td>
<td>17</td>
<td>20.32</td>
<td>4.15</td>
</tr>
</tbody>
</table>
5 Conclusions and future works

In this paper, we explicitly presented an improved solution to the FSSP for tree-based P systems. We improved our previous FSP algorithm [5] by allowing the general to delegate a more central cell in the tree structure, as an alternative to itself, to send the final “firing” command. This procedure for trees-based P systems was extended to digraph-based P systems. Here we use a virtual spanning BFS tree (rooted at the general) in the digraph and use our tree-based middle-cell algorithm for that tree to improve the synchronization time. Alternatively, we would like to develop a way to compute a center of an arbitrary graph since the radius of the graph may be less than the radius of a particular BFS spanning tree. Thus this future work may possibly provide even more guaranteed improvements in synchronization time.

We summarize our work as follows. With $e$ being the eccentricity of the general and $r$ denoting the radius of the graph, where $e/2 \leq r \leq e$, we note the radius $r'$ of the spanning BFS tree satisfies $e/2 \leq r \leq r' \leq e$. Thus, we have the following results:

- If the membrane structure of a considered P system is a tree, then synchronization time is $e + 2r + 3$.
- If the membrane structure of a considered P system is a digraph, then synchronization time $t$ is $e + 2r + 7 \leq t \leq 3e + 7$.

Our empirical work shows that the radius of the BFS spanning tree is often as small as the radius of its host graph and we expect, more often than not, the synchronization time to be closer to $e + 2r + 7$ than to $3e + 7$ for arbitrary digraph-based P systems.

Finally, we mention a couple open problems for the future. We would like to present a theoretical proof based on properties of random trees of why it seems that our gain in performance is independent of the order of the trees considered. The current FSSP solution is designed for digraph-based P systems with duplex channels. Another remaining open problem is to obtain an efficient FSSP solution that synchronizes strongly connected digraphs using simplex channels.

Acknowledgments

The authors wish to thank Ionuț-Mihai Niculescu for providing us with some early empirical statistics on random graphs and to acknowledge the University of Auckland FRDF grant 9843/3626216 to assist our research.

References

P systems with active membranes operating under minimal parallelism

Pierluigi Frisco & Gordon Govan
School of Math. and Comp. Sciences,
Heriot-Watt University,
EH14 4AS Edinburgh, UK,
{P.Frisco, gmg8}@hw.ac.uk

Abstract. We prove that P systems with active membranes operating under minimal parallelism are able to solve NP-complete and PP-complete problems in linear time and exponential space when using different types of rules. We also prove that these systems can simulate register machines.

1 Introduction

Membrane systems (P systems) are an abstract model of computation inspired by the compartmentalisation present in eukariotic cells [10, 11, 3, 12]. P systems can be used as a parallel model of computation, leveraging the inherent parallelism that is present in the cell.

P systems with active membranes are able to compute by using a set of rules that describe the movement and evolution of symbols and compartments. Symbols can move in and out of compartments and evolve into other symbols. Compartments can be divided into more compartments, have their polarity (the charge of the membrane) changed, etc.

Due to their parallelism, P systems can solve NP-complete problems in linear time, but they have to use an exponentially large amount of space to solve such problems, as they try all possible solutions to a problem at the same time.

There are different operating modes in which a P system can run. In this paper we study P systems operating under minimal parallelism as it is less restrictive in its operation than the more commonly used maximal parallelism.

Several results have already been proven when using P systems operating under minimal parallelism, involving computational efficiency and computational completeness.

In [1] it was proved that these systems are capable of solving NP-complete problems in linear time when membrane polarities are allowed. In [5] there are proofs in which NP-complete problems are solved without membrane polarities, but with non-elementary membrane division, as well as a proof without either non-elementary membrane divisions or membrane polarity, but using cooperative evolution rules. We add to these results by solving NP-complete problems in linear time using neither membrane polarities, non-elementary membrane divisions or cooperative evolution rules instead relying on membrane creation.
PP-complete problems were solved by P systems using maximal parallelism in [4]. In this paper we add to this result by again developing a solution to the SQRT-k-SAT problem, but in a minimally parallel manner.

In [1] it was proved that P systems with membrane polarities operating under minimal parallelism are able to simulate register machines. In [5] there are several proofs that do not rely on membrane polarities for simulating register machines but instead requires the use of either promoters, cooperative evolution rules, or label rewriting. We add to these results by allowing the use of creation and dissolution of membranes to simulate register machine.

In the next section we give some definitions relevant to this paper, in Section 3 we discuss different operational modes, in Section 4 we prove how NP-complete and PP-complete problems can be solved by the considered P systems, in Section 5 we prove that a P system can act as a register machine. We end with some comments in Section 6.

2 Basic definitions

We assume the reader to be familiar with Theoretical Computer Science [6], in particular with Membrane Computing [3, 12] and Computational Complexity Theory [9]. Here we only introduce concepts strictly relevant to what is to be presented.

Definition 1. A recognising P system with active membranes is a construct

\[ \Pi = (V, \mu, L_1, \ldots, L_m, R) \]

where:

- \( V \) is an alphabet;
- \( \mu = (W, E, \text{pol}) \) is a vertex-labelled cell-tree (where a cell-tree is a rooted tree such that the root has only one child) underlying \( \Pi \) where:
  - \( W = \{0, 1, \ldots, m\} \) contains vertices. The root of the cell-tree is vertex 0 and each vertex in \( W \) defines a compartment of \( \Pi \);
  - \( E \subseteq W \times W \) defines edges between vertices denoted by \((i, j)\), \( i, j \in W \), \( i \neq j \);
  - \( \text{pol} : W \to \{+,-,0\} \) is a labelling function, called polarity function, associating elements of \( WP = \{+,-,0\} \) to the vertices.
- In the following, \( \mu \) is depicted using the bracket representation common to P systems;
- \( L_i : V \to \mathbb{N}, i \in W \setminus \{0\} \) are multisets of symbols associated with the vertices of \( W \);
- \( R \) is a finite set of rules of the form:
  \[(a) \ [i \rightarrow v \rightarrow w]^p \] 
  with \( i \in W \); \( p \in WP \), \( v \in V \) and \( w : V \to \mathbb{N} \).

Informally: productions associated with compartments;
(b) \( v_i \stackrel{p}{\longrightarrow} w \in W, p, p' \in WP, v \in V \) and \( w \in V \cup \{\epsilon\} \).
Informally: a symbol can pass into a compartment;

(c) \( v_i \stackrel{p}{\longrightarrow} v_i \in W, p, p' \in WP, v \in V \) and \( w \in V \cup \{\epsilon\} \).
Informally: a symbol can pass outside a compartment;

(d) \( v_i \stackrel{p}{\longrightarrow} w \in W \setminus \{0, 1\}, p \in WP, v \in V \) and \( w \in V \cup \{\epsilon\} \).
Short name: dissolution.
Informally: the presence of a symbol in a compartment lets the membrane
defining the compartment to disappear;

(e) \( v_i \stackrel{p}{\longrightarrow} [v, w]_p \in W \setminus \{0, 1\}, p, p', p'' \in WP, v \in V \) and \( w, z \in V \cup \{\epsilon\} \).
Short name: division of a (simple or composite) compartment.
Informally: the presence of a symbol in a compartment lets the compartment
to duplicate;

(f) \( v_i \stackrel{p}{\longrightarrow} v_i \in W \setminus \{0, 1\}, p \in WP, v \in V \) and \( w \in V \cup \{\epsilon\} \).
Short name: creation of a compartment.
Informally: the presence of a symbol in a compartment lets a new compartment
to be created.

Vertex 0 identifies the environment, while vertex 1 identifies the skin compartment of \( \mu \). The set \( W_P = \{+, -, 0\} \) is called set of polarities and its elements are called polarities. Specifically, + is called positive, – is called negative and 0 is called neutral polarity. A compartment having a positive polarity is said to be a positive compartment, similarly for the remaining polarities.

The just given definition of a recognising P system with active membranes is not complete: we only defined the kinds of rules we are going to use in the following, we did not define what a configuration is, etc. For a complete definition of these P systems the reader can refer to [3, 12].

A recognising P system with active membranes II is such that yes, no \( \in V \) and it is said to accept if there is at least one computation such that the symbol yes passes to the environment in the last transition of a finite computation. Such sequences of configurations are called accepting computations. The system II is said to reject the input if for all sequences of configurations the symbol no passes to the environment in the last transition of a finite computation. Such sequences are called rejecting computations.

In this paper we describe P systems solving instances of problems belonging to known complexity classes. For this reason we now introduce the problems we consider and the complexity classes they belong to.

**Definition 2.** Let us consider a set of Boolean variables \( \{v_1, \ldots, v_n\} \) and Boolean
formula \( \psi \) written in conjunctive normal form, that is, \( \psi = C_1 \land \cdots \land C_m \). Each
\( C_j, 1 \leq j \leq m, \) is a clause written as a disjunction of literals \( C_j = l_{j_1} \lor \cdots \lor l_{j_k} \),
each literal being a variable \( v_i \) or its negation \( \bar{v_i} \), \( 1 \leq i \leq n \).
The SAT problem is: given such a Boolean formula ψ is there an assignment of the variables such that ψ is satisfied?

It is known that if \( k_j \geq 3, 1 \leq j \leq m \), then the problem raised by the previous question is \( \text{NP-complete} \) identified by \( k\)-CNF or \( k\)-SAT with \( k \geq 3 \).

**Definition 3.** Let us consider a set of Boolean variables \( \{v_1, \ldots, v_n\} \) and Boolean formula \( \psi \) written in conjunctive normal form, that is, \( \psi = C_1 \land \cdots \land C_m \). Each \( C_j, 1 \leq j \leq m \), is a clause written as a disjunction of literals \( C_j = l_{j_1} \lor \cdots \lor l_{j_{k_j}}, \) each literal being a variable \( v_i \) or its negation \( \bar{v}_i \), \( 1 \leq i \leq n \).

The SQRT-\( k \)-SAT problem is: given such a Boolean formula \( \psi \) are there at least \( \sqrt{2^n} \) assignments verifying \( \psi \)?

It is known that if \( k_j \geq 3, 1 \leq j \leq m \), then the problem raised by the previous question is a \( \text{PP-complete} \) problem identified by \( \text{SQRT-}k\)-SAT with \( k \geq 3 \).

It is also known that \( \text{NP} \subseteq \text{PP} \) [9].

### 3 Operational modes in P systems

P systems can operate in different ways, where by *operate* we mean the way rules are applied. The operational mode of a P system is detached from the definition of the system itself. This means that given a P system, it can operate in different ways. It is known [3, 2] that the operational mode is quite influential in P systems. For instance, a specific P system can have different computational power depending on its type of operation.

Here we list and briefly describe some operational modes relevant for the following sections:

- **asynchronous**: in each transition at least one rule is applied. It is allowed in any transition and for any rule involving a membrane to be applied more than once (as in maximal parallelism);
- **minimal parallelism**: in each transition and for each membrane, if possible, at least one rule is applied. So, this operational mode has to have at least one rule applied for a membrane where one or more rules can be applied. If in a configuration there are different sets of rules that can be applied, then one of them is non-deterministically chosen.
- **maximal strategy**: in each transition any rule that can be applied is actually applied at most once. If in a configuration there are different sets of rules that can be applied, then one of them is non-deterministically chosen. It is important to notice that this operational mode does not aim to maximise the number of applied rules;
- **maximal parallelism**: this is the classical (most used) operational mode in P systems. Similar to maximal strategy but in each transition any rule can be applied more than once. So, in this operational mode it is possible to have transitions with more than one applicable set of rules where some of these sets have the same rules but applied a different number of times.
It is important to notice that in P systems two equal rules present on different membranes are regarded as two different rules. This is independent from the membrane name. This fact is particularly important in P systems with active membranes because (by the application of rules of type $e$) a membrane can be duplicated into two membranes with the same name. So, when such a system computes, the application of the same rule on different membranes with the same name is regarded as the application of two different rules.

In this paper we prove results using P systems operating under minimal parallelism. Such systems were introduced in [1] as a less restrictive model of operation for P systems than both maximal parallelism and maximal strategy. It has been proven that these systems are able to solve NP-complete problems in linear time, but using exponential space to do so. These results can be expanded on by using fewer or different types of rules, and by tackling problems in other complexity classes.

4 Solving NP-complete and PP-complete problems

In this section we show how some families of P systems with active membranes can solve NP-complete and PP-complete problems. As indicated in the following, some of the results in this section improve some results present in the literature of Membrane Computing.

Using the terminology common to Membrane Computing [13], the results present in this section are about P systems constructed in a semi-uniform way (given for a given instance of the problem) and working in a weakly confluent manner (the systems are non-deterministic in their computation, all systems halt and give the same answer independently from the non-deterministic computation).

Theorem 5 in [1] states that recognising non-deterministic P systems with active membranes operating under minimal parallelism, using only rules of type $(a)$-$(e)$ (where rules of type $(e)$ duplicate non-elementary compartments) and polarities can solve $k$-SAT, $k \geq 3$. Our first result, in the next theorem, consists in having a non-deterministic system, not using polarities and using only rules of the kind $(a), (b), (c), (e)$ (duplicating only elementary compartments) and $(g)$.

**Theorem 1.** $k$-SAT, $k \geq 3$, can be solved by recognising P systems with active membranes operating under minimal parallelism without polarities, and using only rules of the kind $(a), (b), (c), (e)$ and $(g)$. The P system requires exponential space and linear time.

**Proof.** Let us consider an instance of $k$-SAT, $k \geq 3$ based on a Boolean formula $\psi$ with $m$ clauses and $n$ variables $\{v_1, \ldots, v_n\}$. Moreover, we define two functions $true$ and $false$, both from $\{v_1, \ldots, v_n\}$ to $\mathcal{P}\{c_1, \ldots, c_m\}$ such that:

- $true(v) = \{c_j \mid \exists r, 1 \leq r \leq k_i \text{ such that } l_{jr} = v\}$
- $false(v) = \{c_j \mid \exists r, 1 \leq r \leq k_i \text{ such that } l_{jr} = \bar{v}\}$

for $v \in \{v_1, \ldots, v_n\}$. So, these functions return the set of clauses verified and falsified by $v$, respectively.
We define the recognising P system with active membranes $\Pi = (V, \mu, L_1, L_2, L_{m+3}, R)$ with:

\[ V = \{ v_i, F_i, T_i \mid 1 \leq i \leq n \} \cup \{ e_j, d_j \mid 1 \leq j \leq m \} \cup \{ c_j, f_j \mid 1 \leq j \leq m+4 \} \cup \{ f, d_{m+1}, \text{yes}, \text{no} \}; \]
\[ \mu = (\{0, 1, 2, m + 3\}, \{(0, 1), (1, 2), (1, m + 3)\}); \]
\[ L_1 = \{ \text{yes} \}; \]
\[ L_2 = \{ v_1 \}; \]
\[ L_{m+3} = \{ e_1 \}; \]

The initial configuration of the system can be constructed in polynomial time by a Turing machine and in a semi-uniform manner. The bracket representation of the initial underlying compartment structure of $\Pi$ is

\[ [0 \ 1 \ f \ [m+3 \ e_1 \ ]_{m+3} \ [2 \ v_1 \ ]_2 \ ]_0 \]

The set $R$ consists of the following rules:

1. $[2 \ v_i \ ]_2 \rightarrow [2 \ F_i \ ]_2 \ [2 \ T_i \ ]_2$ for $1 \leq i \leq n$,
2. $[2 \ F_i \rightarrow \text{false}(v_i)v_{i+1} \ ]_2$ for $1 \leq i \leq n - 1$,
3. $[2 \ T_i \rightarrow \text{true}(v_i)v_{i+1} \ ]_2$ for $1 \leq i \leq n - 1$,
4. $[2 \ F_n \rightarrow \text{false}(v_n)d_1 \ ]_2$
5. $[2 \ T_n \rightarrow \text{true}(v_n)d_1 \ ]_2$
6. $[2 \ d_i \ ]_2 \rightarrow [2 \ [i+2] \ ]_{i+2}$ for $1 \leq i \leq m$,
7. $c_i[i+2 \ ]_{i+2} \rightarrow [i+2 \ d_{i+1} \ ]_{i+2}$ for $1 \leq i \leq m$,
8. $[i+2 \ d_{i+1} \ ]_{i+2} \rightarrow [i+2 \ ]_{i+2} \ d_{i+1}$ for $1 \leq i \leq m$,
9. $[2 \ d_{m+1} \ ]_2 \rightarrow [2 \ ]_2 \ d_{m+1}$
10. $[1 \ \text{yes} \ ]_1 \rightarrow [1 \ ]_1 \ \text{yes}$
11. $[m+3 \ e_1 \rightarrow \epsilon_{i+1} \ ]_{m+3}$ for $1 \leq i \leq 2n + 2m + mn + 4$,
12. $[1 \ d_{m+1} \ ]_1 \rightarrow [1 \ [m+4 \ ]_{m+4} \ ]_1$
13. $f \ [m+4 \ ]_{m+4} \rightarrow [m+4 \ \text{yes} \ ]_{m+4}$,
14. $[m+4 \ \text{yes} \ ]_{m+4} \rightarrow [m+4 \ [m+4 \ ]_{m+4} \ \text{yes}$
15. $[m+3 \ e_k \ [m+3 \ ]_{m+3} \ \epsilon_k \ [m+3 \ ]_{m+3} \ \epsilon_k$ for $k = 2n + 2m + mn + 4$,
16. $[1 \ \epsilon_k \ ]_1 \rightarrow [1 \ [m+5 \ ]_{m+5} \ ]_1$ for $k = 2n + 2m + mn + 5$
17. $f \ [m+5 \ ]_{m+5} \rightarrow [m+5 \ \text{no} \ ]_{m+5}$
18. $[\alpha \ \text{no} \ ]_\alpha \rightarrow [\alpha \ ]_\alpha \ \text{no}$ for $\alpha \in \{m + 5, 1\}$.

In order to facilitate the explanation (groups of) rules have been numbered. In compartment 2 the truth assignments for the $n$ variables are generated, this is performed by the rules 1, . . . , 5. Compartment 2 alternates between its division and the introduction of symbols. During division a value ($T$ for true and $F$ for false) is associated with variable $v_i$. Symbols are introduced by the true and false functions that use up the $T_i$ and $F_i$ symbols and return the $c_i$ symbols associated with the clauses satisfied by $v_i$ and $\bar{v}_i$. After $2n$ transitions all the $2^n$ truth assignments for $\psi$ are generated and present in compartments 2 (called 2-named compartment, in the following). It is important to notice that the application of
rule 1 sees the duplication of elementary compartments (compartments with no other compartments inside).

When rules 4 and 5 are applied in a 2-named compartment, then the introduction of $d_1$ primes the counting of the clauses satisfied by the assignment of variables that the 2-named compartment represents. This counting takes place by the application of rules 6, 7, and 8. Rule 6 uses a $d_i$ to create a compartment with the name $i + 2$, rule 7 will then allow a $c_i$ to pass into that compartment evolving into a $d_{i+1}$. If there is no $c_i$ present it is because clause $i$ is not satisfied by the truth assignment associated to that 2-named compartment and a $d_{m+1}$ will never be generated in this compartment. Rule 8 will then allow the $d_{i+1}$ to pass back into the 2-named compartment, which will then allow rule 6 to be applied again, until $i > m$.

If the truth assignment associated to a 2-named compartment satisfies $\psi$, then, sooner or later, $d_{m+1}$ is present in this 2-named compartment.

After 2n transitions all the assignments are created at most $2m + mn$ transitions are needed for the counting the satisfied clauses (we discuss the $2m + mn$ in a while):

- if a $d_{m+1}$ is present, then rule 9 lets it pass to the skin compartment, then rule 12 lets $d_{m+1}$ create compartment $m + 4$.

If in a transition at least one rule for a membrane is applied, then minimal parallelism allows other applicable rules not to be applied. This means that in the above rules 6, 7, and 8 can take between $3m$ and $2n + mn$ transitions to complete (assuming that they will produce a $d_{m+1}$). Let us assume that in a 2-named compartment the first two clauses are satisfied by the assignment of every variable, meaning there are $n$ copies of both $c_1$ and $c_2$. Rule 6 uses the $d_1$ in the compartment to create a compartment named $i + 2$ (here $i = 1$). Rule 7 is then applied and a $c_1$ then passes into the $i + 2$-named compartment and becomes a $d_2$. As minimal parallelism can allow for just one rule to be applied for each membrane where more rules can be applied, then the next step for the system may either be for the $i + 2$ membrane to have another $c_1$ pass in, or to pass the $d_2$ out. In the worst case (as we have assumed here) there will be $n$ copies of $c_i$ that may pass into the $i + 2$ membrane before a $d_2$ is allowed to pass out, meaning the longest time it could take for these rules to be applied would be $2 + n$ steps.

As there are $n$ copies of $c_1$ then $n$ copies of $d_2$ may be made. In this case $n$ copies of a $i + 2$ membrane may be created (now $i = 2$), which may take at most $n$ steps, but as we are using minimal parallelism rules 6 and 7 and rules 6 and 8 can be applied at the same time as the $i + 2$ membranes can be different. However, this means that rule 8 may pass out a $d_3$ into the 2-named compartment before rule 6 has applied to all the copies of $d_2$ meaning that the 2-named compartment may perform rule 6 on all copies of $d_2$ before performing it on a $d_3$. This means it may be a further $n$ steps before a $i + 2$ compartment is made, where now $i = 3$, so all three rules again take $2 + n$ steps to complete. Following this logic it is a simple leap to see that no matter the number of copies
of $d_i$ and $c_i$, each of $m$ clauses can be counted in at most $2 + n$ steps, giving at most $2m + mn$ steps to count all the clauses. The possibility to chose what rule to apply lets $\Pi$ be non-deterministic.

The above means that if at least one assignment verifies $\psi$, then in the worst case after $2n + 2m + mn + 3$ transitions there will be several $m + 4$ compartments in compartment 1. When this happens, then in transition $2n + 2m + mn + 4$ minimal parallelism forces object $f$ to pass into one of these $m + 4$ compartments and while doing so it becomes \textbf{yes} (rule 13). The application of rules 10 and 14 allow \textbf{yes} to pass to the environment. When this happens the system halts.

While what just described goes on, then also rule 11 is applied. This rule lets the object $e$ increase its subscript

If no assignment verifies $\psi$, then after $2n + 2m + mn + 4$ transitions, then rule 15 is applied letting $e_{2n+2m+mn+4}$ pass to the skin compartment. Then rule 16 lets $e_{2n+2m+mn+4}$ create compartment $m + 5$ in the skin compartment. When this happens minimal parallelism forces $f$ to pass into this compartment while becoming \textbf{no} (rule 17). The application of rule 18 lets \textbf{no} to pass to the environment. When this happens the system halts.

Notice that as there is only one $f$ in the skin compartment, then if at least one assignment verifies $\psi$ this $f$ will pass to one of the $m + 4$ compartment before it can pass to the $m + 5$ compartment created by the application of rule 16.

It should be clear that the statement holds if minimal parallelism is replaced by maximal strategy or maximal parallelism. If maximal strategy is in place, then as rules that can be used must be used once per step then the system would run in much the same manner as it would using minimal parallelism, there would be a few differences in how rules 7 and 8 and rules 6 and 9 work together, but it would still take at most $2n + 2m + mn + 1$ steps to pass a $d_{m+1}$ into the skin compartment and perform in a non-deterministic manner. If maximal parallelism is in place, then in the worst case the system would behave as in maximal strategy, using each rule at most once in every transition, but it is also able to perform rules multiple times in each transition, and would be able to pass a $d_{m+1}$ into the skin compartment in transition $2n + 3m + 1$.

Clearly $\Pi$ uses linear time and exponential with respect to $mn$. $\square$

P systems operating under maximal parallelism are known to be able to solve problems in the \textbf{PP-complete} complexity class [7]. The \textbf{PP-complete} complexity class involves counting the number of valid solutions that a \textbf{NP-complete} problem has. Theorem 2 finds a solution to SQRT-$k$-SAT, an example of a \textbf{PP-complete} problem that can be solved by the considered P systems and has previously been solved using maximal parallelism in [4].

\textbf{Theorem 2.} SQRT-$k$-SAT is a problem that is satisfied if $\sqrt{2^n}$ of $2^n$ possible truth assignments satisfy a given $k$-SAT problem. SQRT-$k$-SAT, $k \geq 3$, can be solved by recognising P systems with active membranes operating under minimal parallelism, with polarities, and using only rules of the kind (a), (b), (c) and (e). The P system requires exponential space and linear time.
Proof. Let us consider an instance of SQRT-k-SAT, k \geq 3 based on a Boolean formula \( \psi \) with \( m \) clauses and \( n \) variables \( \{v_1, \ldots, v_n\} \). We use the functions \textit{true} and \textit{false} that we defined in the proof of Theorem 1.

The construction of the P system is dependent on the value of \( n \). The system has to accept a \( \psi \) with \( \sqrt{2^n} \) satisfying variable assignments. To correctly count \( \sqrt{2^n} \) we first consider the binary form of \( \lceil \sqrt{2^n} \rceil - 1 \) (we want the system to accept if there are \( \lceil \sqrt{2^n} \rceil \) satisfying variable assignments). In binary form of \( \lceil \sqrt{2^n} \rceil - 1 \) and for each \( j = 0, \ldots, \lceil \log(\lceil \sqrt{2^n} \rceil - 1) \rceil \), if the \( j \)-th least significant bit is 1, then we create a compartment numbered \( k + 3 + j \) and put a \( p_j \) symbol inside it.

These compartments will then be divided into \( \lceil \sqrt{2^n} \rceil - 1 \) compartments once the system is running.

We define the recognising P system with active membranes \( P = (V, \mu, L_1, \ldots, L_{k+3+l}, R) \) with:

\[
V = \{v_i, F_i, T_i \mid 1 \leq i \leq n\} \cup \{c_j, d_j \mid 1 \leq j \leq m\} \cup \{p_i \mid 0 \leq i \leq l\}
\cup \{s_i \mid 1 \leq i \leq 2n + 2m + 3\} \cup \{d_{m+1}, f, a, r, \text{yes, no}\};
\]

\[
L_1 = \{f\};
\]

\[
L_2 = \{v_i\};
\]

\[
L_i = \{v_i\};
\]

\[
L_k = \{s_i, a, r\};
\]

\[
L_{k+1} = \emptyset
\]

\[
L_{k+i} = \emptyset
\]

\[
L_{k+3+j} = p_j, 0 \leq j \leq l
\]

where these compartments exist as described above where \( k = m + 3 \) and \( l = \lceil \log(\lceil \sqrt{2^n} \rceil - 1) \rceil \).

The initial configuration of the system can be constructed in polynomial time by a Turing machine and in a semi-uniform manner. The bracket representation of the initial underlying compartment structure of \( P \) is

\[
[0 \mid 1 \ f \ \ [k+3 \ p_0]_{k+3} \ \cdots \ [k+3+i \ p_i]_{k+3} \ [k \ s_i, a, r]_{k} \ [k+1]_{k+1} \ [k+2]_{k+2} \ [2 \ v_1 \ [3]_{3} \ \cdots \ [2+m]_{2+m} \ [1]_{1} \ [0]]_{0}
\]

The set \( R \) consists of the following rules:

1. \[ [2 \ v_i]_{i} \to [2 \ F_i]_{i} \ [2 \ T_i]_{i} \ \text{for} \ 1 \leq i \leq n, \]
2. \[ [2 \ F_i \to \text{false}](v_i)_{i+1} \ [2]_{i+1} \ \text{for} \ 1 \leq i \leq n - 1, \]
3. \[ [2 \ T_i \to \text{true}(v_i)_{i+1} \ [2]_{i+1} \ \text{for} \ 1 \leq i \leq n - 1, \]
4. \[ [2 \ F_n \to \text{false}(v_n)d_{1} \ [2]_{2} \]
5. \[ [2 \ T_n \to \text{true}(v_n)d_{1} \ [2]_{2} \]
6. \[ [2]_{2+1} \ [2]_{i+1} \to [2+1]_{2+1} \ [2]_{i+1} \ \text{for} \ 1 \leq i \leq m, \]
7. \[ [2+1]_{2+1} \ [2+1]_{i+1} \to [2+1]_{2+1} \ [2+1]_{i+1} \ \text{for} \ 1 \leq i \leq m, \]
8. \[ [2+1]_{2} \ [2]_{i+2} \ [2]_{i+2} \ [2]_{i+1} \ \text{for} \ 1 \leq i \leq m, \]
9. \[ [2]_{m+1} \ [2]_{m} \ \text{for} \ \text{for} \ 0 \leq j \leq l, \]
10. \[ [k+3+j \ p_i]_{k+3} \ [k+3+j \ p_{i-1}]_{k+3+j} \ [k+3+j \ p_{i-1}]_{k+3+j} \ [0]_{i} \ \text{for} \ 0 \leq j \leq l, \]

\[ \text{for} \ 1 \leq i \leq l \]
In order to facilitate the explanation (groups of) rules have been numbered.

In compartment 2 the truth assignments for the \( n \) variables are generated, this is performed by the rules 1, ... , 5. Compartment 2 alternates between its division and the introduction of symbols. During division a value (\( T \) for true and \( F \) for false) is associated with variable \( v_i \). Symbols are introduced by the \textit{true} and \textit{false} functions that use up the \( T_i \) and \( F_i \) symbols and return the \( c_i \) symbols associated with the clauses satisfied by \( v_i \) and \( \bar{v}_i \). After 2n transitions all the \( 2^n \) truth assignments for \( \psi \) are generated and present in compartments 2 (called \textit{2-named compartment}, in the following). It is important to notice that the application of rule 1 sees the duplication of non-elementary compartments (compartments with other compartments inside).

Every time a \( c_i \), \( 1 \leq i \leq n \), is generated, then the application of rule 6 lets this symbol enter compartment \( 2+i \) and lets this compartment change polarity. This means that in each 2-named compartment, rule 6 for a particular \( i \) can be applied at most once.

When rules 4 and 5 are applied in a 2-named compartment, then the introduction of \( d_i \) primes the counting of the compartments that were given a positive polarity by rule 6 in that 2-named compartment. This counting takes place by the application of rules 7 and 8. It should be clear that the \( d_i \) symbol cannot visit the same \( i + 2 \)-named, \( 1 \leq i \leq n \), compartment more than once. This is because once such a compartment has been visited by \( d_i \) (rule 7), then the application of rule 8 lets the polarity of that compartment change such that rule 7 cannot be applied again for a specific \( i \). It should also be clear that if an \( i + 2 \)-named compartment with a positive membrane polarity is not present then the truth assignment associated to that 2-named compartment does not satisfy clause \( i \) and \( d_{i+1} \) cannot be generated in that 2-named compartment.
If the truth assignment associated to a 2-named compartment satisfies $\psi$, then, sooner or later, $d_{m+1}$ is present in this 2-named compartment. When this happens rule 9 is applied such that a $d_{m+1}$ is passed into the 1 compartment. A $d_{m+1}$ is passed back into the 1 compartment for each truth assignment that validates $\psi$.

The compartments $k + 3 + j, k = n + 3, 0 \leq j \leq \lfloor \log([\sqrt{2^7}] - 1)\rfloor$, present in the current system are used to count the number of assignments satisfying $\psi$. If each of the compartments receives a $d_{m+1}$ and there is at least one extra $d_{m+1}$, then yes will pass to the environment.

We describe how the count of the assignments satisfying $\psi$ takes place in $\Pi$ with the help of an example. The generalisation to any computations of $\Pi$ should be straightforward.

The search for assignments satisfying $\psi$ goes on in the 2-named compartments and afterwards $d_{m+1}$ symbols are present in compartment 1. Consider for example that $n = 7$, so that there are $2^7 = 128$ assignments. $\Pi$ lets yes pass to the environment only if at least $11 + z, z \geq 1$, assignments verify $\psi$, as $\lfloor \sqrt{2^7} \rfloor - 1 = 11$. If 11 or fewer assignments verify $\psi$, then yes will never pass to the environment, and no will be passed out instead. So, it is a matter to count the number of assignments. This counting has to consider that the number of $d_{m+1}$ is an exponential function of the input. This means that the counting of the $d_{m+1}$ has to be done in an efficient way: it cannot be done in a ‘linear’ way, one after the other, because in this case the best time complexity would be exponential.

Let us assume that there are $11 + z, z \geq 1$, assignments verifying $\psi$. 11 of these assignments are counted in an efficient way. As $n = 7$ then $\lfloor \sqrt{2^7} \rfloor - 1 = 11$ and the bit string for 11 is 1011, so initially compartment 1 contains: $[k+3]_0^0 [k+3]$, $[k+4]_1^0 [k+4]$, and $[k+6]_3^0 [k+6]$. Rule 10 lets these compartments duplicate such that at the end of this process:

- 8 copies of $[k+6]_0^0 [k+6]$
- 2 copies of $[k+4]_1^0 [k+4]$
- 1 copy of $[k+3]_0^0 [k+3]$

are present, giving a total of 11 compartments numbered between $k + 3$ and $k + 3 + \lfloor \log([\sqrt{2^7}] - 1)\rfloor$ inclusive. These compartments perform the counting.

Each of these 11 compartments are able to have a $d_{m+1}$ pass into them according to rule 11. As we are using minimal parallelism and there are 11 separate membranes, then one copy of $d_{m+1}$ can pass into each of them and change the polarities to positive all in a single transition.

As after this there is still at least one copy of $d_{m+1}$ still in compartment 1 then we wish to be able to pass a yes to the environment. In the transition after the counting is performed an a symbol (we will explain where it has come from shortly) passes into the $k + 1$ compartment according to rule 14, making the polarity positive. Now that $k + 1$ has a positive polarity, rule 15 will allow it to admit a $d_{m+1}$, this changes the polarity to negative so that no other copies of $d_{m+1}$ may enter. Rule 16 lets the $f$ symbol pass into the $k + 1$ compartment.
and to become a yes. Rules 17 and 18 then allow the yes to pass into the environment.

While all of the above is happening, in parallel rule 12 is also being used in the k compartment. This is allowed when operating under minimal parallelism, as it is the only rule being used in the compartment.

Rule 12 increments the s_i object from s_{i1} to s_{i2n+2m+3}. This is the same number of steps required to ensure that the rest of \Pi has finished evolving and counting the copies of d_{m+1}. Rule 13 then lets the a symbol pass out of the k compartment, changing the polarity to positive. If there are any free copies of d_{m+1} left in compartment 1 then the a is used as described above. In the next transition the r symbol then passes out of the k compartment according to rule 19. Rule 20 then allows the r to pass into the k + 2 compartment giving it a positive polarity. Rule 21 then allows the r to pass out of the k + 2 compartment, changing the polarity to negative. Rule 22 will then allow the f symbol to enter the k + 2 compartment, changing to a no symbol, but this will only happen if the f symbol has not already entered the k + 1 compartment, which it would have done if there were any free copies of d_{m+1} left in compartment 1. Therefore the no symbol is only produced if there were not at least \sqrt{2^n} assignments satisfying \psi. Rules 23 and 24 will then allow no to pass into the environment.

There being only one f symbol in the system guarantees that only one of yes or no will pass out into the environment and halt the system.

Like the proof for k-SAT in Theorem 1 this \Pi runs in linear time in respect to n + m and requires exponential space to check all possible assignments of values. \Pi would also be valid solution to this problem if a maximal strategy or maximal parallelism operating mode was used.

5 Computational power

In [1] it is proven that the class of numbers generated by P system with active membranes with polarities, using only rules of type (a), (b) and (c) and operating under minimal parallelism is equivalent to the class of numbers generated by register machines. In the following we prove that in these systems the presence of polarities can be replaced by rules of type (d) and (g).

Register machines are composed of registers (also called counters) each storing a natural number [8]. Simple operations can be performed on the registers: addition of one unit and conditional subtraction of one unit. After each of these operations the machine can change state.

A register machine with n registers, n \in \mathbb{N}_+, is formally defined as M = (S, I, s_1, s_f), where S = \{s_1, \ldots, s_f\} is a finite set of states, s_1, s_f \in S are respectively called the initial and final states, I is the finite set of instructions of the form (s, \gamma_i, v, w) or (s, \gamma_i^+, v) with s, v, w \in S, s \neq s_f, 1 \leq i \leq n.

A configuration of a register machine M with n registers is given by an element in the n+1-tuples S \times \mathbb{N}_0^n. Given two configurations (s, val(\gamma_1), \ldots, val(\gamma_n)), (s', val(\gamma_1'), \ldots, val(\gamma_n')) (where val : {\gamma_1, \ldots, \gamma_n} \rightarrow \mathbb{N}) is the function returning
the content of a register) we define a computational step as \((s, \text{val}(\gamma_1), \ldots, \text{val}(\gamma_n)) \vdash (s', \text{val}(\gamma'_1), \ldots, \text{val}(\gamma'_n))\) and:

if \((s, \gamma_i^-, v, w) \in I\) and \(\text{val}(\gamma_i) \neq 0\), then \(s' = v, \gamma'_i = \text{val}(\gamma_i) - 1, \gamma'_j = \text{val}(\gamma_j), j \neq i, 1 \leq j \leq n\);
if \(\text{val}(\gamma_i) = 0\), then \(s' = w, \gamma'_j = \text{val}(\gamma_j), 1 \leq j \leq n\);
(informally: in state \(s\) if the content of register \(\gamma_i\) is greater than 0, then subtract 1 from that register and change state into \(v\), otherwise change state into \(w\))

if \((s, \gamma_i^+, v) \in I\), then \(s' = v, \gamma'_i = \text{val}(\gamma_i) + 1, \gamma'_j = \text{val}(\gamma_j), j \neq i, 1 \leq j \leq n\);
(informally: in state \(s\) add 1 to register \(\gamma_i\) and change state into \(v\)).

**Theorem 3.** The class of numbers generated (accepted) by \(P\) system with active membranes operating under minimal parallelism is equivalent to the one of register machines.

**Proof.** Let \(M = (S, I, s_1, s_f)\) be an accepting register machine with \(n\) registers \(\gamma_i, 1 \leq i \leq n\), whose initial value is \(\text{val}(\gamma_i), 1 \leq i \leq n\), respectively. We describe a \(P\) system with active membranes \(Π = \langle V, µ, L_1, R \rangle\) as in the statement simulating \(M\) (as the proof for a generating register machine is similar to the given one, it is not given).

The system \(Π\) is such that:

\[
V = S \cup \{c_i, d_i \mid 1 \leq i \leq n\} \cup \{s', s'', s'''', v', t_{i,j}, t''_{i,j}, e_{i,j}, g_{i,j} \mid j : (s, \gamma_i^-, v, w) \in I\};
L_1 = \{s_1\};
\]

where, for the sake of simplicity, we assume the instructions in \(I\) to be uniquely numbered. The underlying cell-tree is better defined by the bracket representation:

\[
[0[1[c_1]c_1 \cdots [c_1]c_1 \cdots [c_n]c_n \cdots [c_n]c_n]0]
\]

where the number of \(c_i\) compartments inside compartment 1 is equal to \(\text{val}(\gamma_i), 1 \leq i \leq n\).

The rules in \(R\) are given in the following.

For each instruction of the kind \((s, \gamma_i^+, v) \in I\) the set \(R\) contains:

1: \([1 \ s \rightarrow c_i]_1\)
2: \([1 \ c_i]_1 \rightarrow [1[c_i] v]_1\)
3: \([c_i] v]_c_i \rightarrow [c_i]_c_i v\)

while for each rule of the kind \((s, \gamma_i^-, v, w) \in I\) the set \(R\) contains:
In order to facilitate the explanation, rules have been numbered. The addition of 1 to register \( \gamma_1 \) in \( M \) is simulated in \( \Pi \) by the creation of a compartment \( c_i \) inside compartment 1. The subtraction of 1 to register \( \gamma_i \) in \( M \) is simulated in \( \Pi \) by the dissolution of a \( c_i \) compartment present in compartment 1. As already said, in the initial configuration \( \Pi \) has as many \( c_i \) compartments inside compartment 1 as \( \text{val}(\gamma_i), 1 \leq i \leq n \). Moreover, \( s_1 \), indicating the simulation of \( M \) being in its initial state, is present in compartment 1. In general, in \( \Pi \) the presence of an element \( s \in S \) in compartment 1 indicates the simulation of \( M \) being in state \( s \). The system \( \Pi \) simulates \( M \) faithfully: if \( \Pi \) starts with \( \text{val}(\gamma_i) \) compartments \( c_i \) in its skin compartment it reaches a halting configuration with \( s_f \) in its skin compartment if and only if \( M \) accepts \( \text{val}(\gamma_i), 1 \leq i \leq n \).

The simulation of instruction of the kind \((s, \gamma_i^+, v)\) is performed by the sequential application of rules 1, 2 and 3. The simulation of instructions of the kind \((s, \gamma_i^-, v, w)\) is a bit more laborious as it can occur by different sequences of transitions. We describe here the shorter sequences and we discuss the other cases later on. In order to do so clearly, we employ the following notation:

rule numbers grouped in parentheses have to / can occur in parallel (because of the operational mode);

a subscript indicates in which compartment a rule is applied (that is, to what set \( R \) the rule belongs to).

If a \( c_i \) compartment is present in compartment 1, then the applied rules are: \( 4_1 (5_1, 12_1, 19_1), 6_1, 13_1, 20_1 \), \( 7_1, 14_1, 8_2, 9_2, 10_1, 11_1 \). As \( \Pi \) operates under minimal parallelism we cannot assume that more than one rule is applied in
a compartment even if this could occur. So, in the previous sequence of applied rules, we cannot assume that \((5_1, 19_1)\) and \((6_1, 20_1)\) are applied in parallel. It could be that only one of the two applicable rules is applied. If this would be the case, \(\Pi\) would still perform the expected simulation by, for instance, one of the following two sequences of applied rules: \((4_1, (5_1, 12_c), (6_1, 13), (7_1, 14_c), 8_g, 9_g, 19_1, 20_1, 10, 11)\) or \((4_1, (12_c, 19_1), (13, 20_1), (5_1, 14_c), 6_i, 7_i, 8_g, 9_g, 10_i, 11_i)\). It is crucial to notice that minimal parallelism forces the application of \(12_c\) together with \(5_1, 19_1\) or both. If this were not the case (under asynchronous operational mode, for instance), then \(\Pi\) would not perform the desired simulation.

If no \(c_i\) compartment is present in compartment 1, then the applied rules are: \((4_1, (5_1, 19_1), (6_1, 20_1), (7_1, 21_t), 15_t, 16_1, 17_c, 18_c)\). Also in this case the simulation would be properly performed even if only one rule per compartment is applied when possible.

6 Final remarks

Classically, computational complexity classes are defined in terms of the time and space needed by Turing machines to solve specific problems. The flexibility present in membrane systems allows to (partially) translate time and space requirements into features (type of rules, presence/absence of polarities, operational mode) allowed in specific P systems.

Theorems 1 and 2 in the present paper fall in this line of research: we prove that the \textbf{NP-complete} and \textbf{PP-complete} problems present can be solved by P systems having different features. Unfortunately, this result does not show a proper inclusion: we do not know if altering any of the features of the P systems considered in these theorems do not allow these systems to solve \textbf{NP-complete} and \textbf{PP-complete} problems. Formally: we proved that these features are sufficient but we did not prove that they are necessary. Proving the necessity of these features would be an important result.

We want to make clear that Theorem 1 does not improve [1][Theorem 5]. Even if the P system in Theorem 1 does not use polarities (as the P system in [1][Theorem 5]) it does use types of rules not present in the P system in [1][Theorem 5].

References


Morphological Algorithms: Membrane Receptor-ligand Interactions and Rule-based Molecule Graph Evolution for Exact Set Cover Problem

Gerd Gruenert\textsuperscript{a1}, Gabi Escuela\textsuperscript{a1}, Peter Dittrich\textsuperscript{a}, and Thomas Hinze\textsuperscript{b}

Friedrich Schiller University Jena
\textsuperscript{a}Department of Computer Science, Bio Systems Analysis Group
\textsuperscript{b}School of Biology and Pharmacy, Department of Bioinformatics
Ernst-Abbe-Platz 1–4, D-07743 Jena, Germany
{gerd.gruenert,gabi.escuela,peter.dittrich,thomas.hinze}@uni-jena.de

Abstract. Spatial structures and their dynamics turned out to be ideal candidates for data storage and information processing. Algorithmic design can unequivocally benefit from exploitation of the features provided by computing in space. The corresponding field, to which we refer to as morphological algorithmics, constitutes a promising branch within membrane computing when considering spatial structures found in biological compartments and conformations of macromolecules. Plasticity of receptors residing on cell membranes as well as agility of transfer RNA containers exhibits two forms of morphology worth to be utilised for computing purposes. To this end, we introduce two novel approaches to solve by morphological algorithms the exact set cover problem known to be NP-complete. Both approaches overcome the insufficiencies caused by the need of a pre-compiled exponential workspace. Moreover, they combine the capability of self-organization with an exhaustive search. We present both algorithms along with consistent simulation case studies carried out using P-Lingua and SRSim.

1 Introduction

Morphology emerged as an interdisciplinary science focused on \textit{shapes}. Shapes comprise one-, two- or three-dimensional spatial structures. In nature, we are surrounded by a variety of different matters, solid materials, and artifacts. The history of mankind reveals numerous examples on how brave pioneers became aware of the importance of shapes fascinated by the idea to employ dedicated matter as a \textit{tool} to fulfill a specific \textit{function}. Stones shaped like a cutting edge appeared to be ideal candidates to divide loaves of meat into smaller pieces while stones with a planar surface properly represent an anvil. The relationship between shape and function opened the mind towards uncountable inspirations for more or less useful devices. The era of mechanics originated the first generation of computing machines based on gear drives. The famous Schickard engine was

\textsuperscript{1} Authors contributed equally.
able to carry out addition, subtraction, multiplication, and division on a fixed number of six digits. Even if nobody would seriously call Schickard’s engine a morphological computer, the structure and spatial adjustment of matter serves both, data storage and data processing. Obviously, the precise design of the system ensures its function.

Alternatively, a successive modification of material shape and surface can emulate a trial-and-error optimisation process corresponding to a non-trivial mathematical task. Imagine the construction of a solid spatial structure aimed to have a minimal aerodynamic resistance. Starting with a cuboid block, its surface could become more and more grinded and hence shaped until the optimal form is reached. Having in mind that this optimal form also represents the solution of an underlying system of ordinary differential equations derived from fluid dynamics, we can state that shaping is capable of solving selected mathematical problems.

Even for universal computation purposes, the usage of shapes in a pure mechanistic sense succeeded in the introduction of a mechanical Turing machine [31]. It is composed of a finite collection of spatial elements whose specific surface structure allows a dedicated assembly of the machine. Its operation is based on the consecutive replacement of its spatial elements.

Nevertheless, the exploitation of morphology to carry out computational tasks is of course no invention of engineering. Various life forms discovered the advantages of shapes employed as helpful tools much earlier. Particularly, the slime mold _Physarum polycephalum_ brings its functional shaping to perfection in order to optimise collection of nutrients from different environmental locations [27]. Physarum growth coincides with path finding in a two-dimensional space. Here, macro-scaled morphology solves the problem to identify the shortest network to connect places (sources of nutrients). This ability of the slime mold could be successfully adopted to construct a Physarum machine able to emulate the behaviour of a Kolmogorov-Uspensky machine, a model of computation known to be Turing complete [1].

On the other hand, there are several examples of molecular solutions of NP-complete problems in the literature. In the famous first DNA Computing paper, Adleman proposed the use of real DNA strands to solve a variant of the Hamiltonian path problem [2], generating first the space of all candidate solutions via hybridisation and ligation. Lipton developed also DNA experiments to solve the SAT problem [25]. Recently, using this approach, a solution of the set problem has been proposed [9]. In related simulation systems _in silico_ we can refer to Banzhaf [5], who proposed an artificial chemistry optimisation approach to the traveling salesman problem. An _in vitro_ evolution of RNA strands in the presence of specific ribozymes (RNase) was successfully employed to tackle the knight-problem [11]. This algorithm advantageously combines the idea of molecular computing with evolutionary elements based on molecular spatial structures.

The paper is organized as follows: in Section 2 we introduce briefly the notion of morphological algorithms. In Section 3 we present a definition of the exact set cover problem, and discuss about related problems, applicability and complexity. We also give two instances of the problem that are considered throughout the
paper. A brute-force solution based on a simple artificial chemistry is proposed
and implemented to solve the simplest instance, using only concentration of
species as information. Sections 4 and 5 contain the design, implementation and
results from two models that use such a morphological approach to solve the
exact set cover problem based on P Systems and rule-based reaction systems,
respectively. Further considerations and implications about the frontiers of the
morphological algorithmic perspective are discussed in Section 6. We conclude
the paper with some final remarks given in Section 7.

2 Morphological Algorithms

Computing by morphology [7] could be the next natural step in programming
with artificial chemistries. In our approach, we abstract the notion of a physical
shape or form in well defined structure-function encapsulated components. The
key point here is to implicitly define molecules using compound objects based
representations [10], as in object-oriented modelling and programming languages.
The compound objects we are interested in are inspired by shapes that can be
found in nature, and whose functionality could be useful to a given problem.
Objects are represented by membranes or (complex) molecules in the artificial
chemistry [10] approaches like membrane computing [29] and rule-based mod-
eling [15, 14].

Therefore, morphological algorithms deal with reactive objects as
data/program elements, that are able to perform structural changes to others or
to itself. The functionality of these objects and the way they interact with their
environment are determined by the current shape configuration. This gives to
each component a certain autonomy that could be exploited to model systems
with no global control. Additionally, this allows to avoid algorithms with separate
phases like generation, calculation, correction, etc., leading to fully concurrent
systems.

Doing computations with such abstractions of biochemical elements that re-
act and adapt to their context could be an advantageous feature in dynamic
optimisation problems [34]. Another benefit is the generation of solution space
on the fly, with no need of comprehensively pre-compiled structures. Moreover,
this concept, related to the object-oriented paradigm from computer sciences,
could be employed to design and help to implement chemical programs in vitro
or in vivo. Although this idea is not new, we consider that our contribution will
help with the design and implementation of such programs, specially considering
the development and integration of practical applications using modelling and
simulation tools.

When migrating from a macro scale to a micro scale, the relation between
morphological algorithmics and membrane computing becomes obvious: Pro-
cesses of molecular assembly like in vitro DNA tiles [33] but also in vivo structures
like molecular motors, transport proteins, or cell surfaces receive their function
by the three-dimensional shape of its interacting parts. Throughout this paper,
we develop and explore various mechanisms of in silico (molecular) assembly in
three-dimensional space to tackle the exact set cover problem. Its major feature lies in the exploitation of shape in terms of the spatial arrangement of molecular structures beyond simple string-based representations.

3 Exact Set Cover

3.1 Problem definition and relevance

We will first give a formal definition for the exact set cover problem, which is NP-complete [22]:

Let \( \mathcal{X} = \{x_1, x_2, \ldots, x_n\} \) a finite set and \( \mathcal{F} = \{S_1, S_2, \ldots, S_k\} \) a family of sets (also called collection), such that \( \forall i \in \{1 \ldots k\}((S_i \neq \emptyset) \land (S_i \subseteq \mathcal{X})) \). Then, \( \mathcal{C} \subseteq \mathcal{F} \) is an exact cover of \( \mathcal{X} \) if and only if:

1. \( \mathcal{X} = \bigcup_{S \in \mathcal{C}} S \), and
2. \( \forall S, S' \in \mathcal{C}, S \neq S', S \cap S' = \emptyset \)

To find all the solutions for this problem, Knuth [23] proposed a so called Algorithm X, a recursive, nondeterministic algorithm that combines depth-first and backtracking as programming strategies, and as data a matrix that represents the relation “contained in” between elements and subsets. Additionally, the author suggests the use of doubly linked circular lists, as they seem to be the most effective data structure when implementing this approach.

The exact set cover problem shares common properties with other NP-complete problems, including those that consider other kinds of cover. In the minimal set cover, the idea is to find the set that covers \( \mathcal{X} \) with the sub-collection of minimum size taken from \( \mathcal{F} \), no matter that the selected subsets overlap. A specially studied case of the exact set cover problem, is to find a minimum size exact cover for \( \mathcal{X} \) using subsets \( S_i \) each of them containing exactly 3 elements [9, 13, 22]. Moreover, other constraint satisfaction problems, as the N-queens problem, Pentomino tilings and Sudoku puzzle are variants of the exact set cover.

Set cover problems, in general, have their practical applications in conservation biology [26], phylogenetics [18] and protein identification [19].

Two instances of the set cover problem are displayed in Figure 1. The search space is growing exponentially with the number of available solution candidates \( C_i \in \mathcal{P}(\mathcal{F}) \) from the power set of the available subsets. Each element \( C_i \) of the power set has to be considered as a potential solution. This means, the chance of randomly generating a correct solution is \( \frac{1}{2^{|\mathcal{F}|}} \), where the notation \( |\mathcal{F}| \) denotes the cardinality of the set \( \mathcal{F} \). This implies a probability of \( \frac{1}{2} = 12.5\% \) in the case displayed in Figure 1(a) and \( \frac{1}{1024} \approx 0.1\% \) in the case of 10 subsets as in Figure 1(b).

Still, for the algorithms presented here, these problem instances are almost too simple, because our algorithms are using structured molecules and membranes that allow us to ignore solution candidates from \( c_i \in \mathcal{P}(\mathcal{F}) \) if they would lead to multiple usage of elements \( x_i \) from \( \mathcal{X} \), i.e. that violate condition 2. from the problem definition given above. Similar to the algorithm by Knuth [23],
we are only generating solutions whose elements do not overlap. This does not
mean, that the problem becomes trivial when using these algorithms - there are
still many combinations of non-overlapping subsets that will not lead to an ideal
solution. Considering the problem from Figure 1(b) for example and including
the subset \( E \), which covers only the element \( a \), it will not be possible any more
to cover the element \( c \) because the subset \( A \) cannot be selected any more, since
now element \( a \) would be covered twice.

![Diagram](image)

**Fig. 1.** Two exemplary instances of the exact set cover problem: Circles on the left of
each subfigure represent the different subsets \( A, B, C, \ldots \) of the elements \( a, b, c, \ldots \) shown
on the right. Arrows leaving a subset indicate all the elements that belong to this subset.
Hence, these arrows are a representation of the externally given problem instance and
cannot be modified by the algorithm. The thick circles on the left of each figure indicate
the perfect set of subsets and thus a solution to the exact set cover task. That is, each
element on the right is reached by exactly one arrow, when exactly the thick subsets
on the left are chosen. In a simple instance of the problem \( X = \{a, b, c, d, e\} \) and \( \mathcal{F} = \{A = \{a, d\}, B = \{a, b\}, C = \{c, d, e\}\} \), the subsets \( B \) and \( C \) lead to the perfect solution,
while \( A \) only covers the elements \( a \) and \( d \) but also disallows the subsets \( B \) and \( C \) to
to cover more elements. (b) In the larger example with \( X = \{a, b, c, d, e, f, g, h, i, j\} \) and
\( \mathcal{F} = \{\{a, b, c\}, \{d, e, f\}, \{g, h\}, \{i, j\}, \{a\}, \{d\}, \{g\}, \{i\}, \{b\}, \{e\}\} \) there are ten elements
as well as ten subsets of elements.
We calculate the probability to obtain a perfect solution through random assembly by constructing the tree of all possible generation pathways, starting from the empty set. Each node in the tree is a solution candidate \( C^{(n)}_i \in \mathcal{P}(\mathcal{F}) \), containing \( n \) subsets. For each node \( C^{(n)}_i \), we add all compatible subsets \( S_k \in \mathcal{F} \), as child nodes \( C^{(n+1)}_j = C^{(n)}_i \cup \{S_k\} \), producing nodes of size \( n + 1 \). Subsets are compatible, when they do not cover overlapping elements:

\[
\left( \bigcup_{S \in C^{(n)}_i} S \right) \cap S_k = \emptyset
\]

and will thus result in new solution candidates conform to condition 2. from the problem definition. Since two subsets \( S_i \) and \( S_j \) can be added in different order, there will be a number of identical solution candidates present in the tree. The proportion of the leaves that produced perfect solutions to the number of leaves with non-perfect solution gives the probability of randomly generating perfect solutions. This leads to the probability of \( \frac{2}{3} \) for the three-subsets problem instance defined in Figure 1(a) and to a chance of 0.0256 in case of Figure 1(b).

3.2 An artificial chemistry for brute-force solution

The exact set cover problem can be solved deterministically using an artificial chemistry in a brute-force manner. A corresponding approach avoids exploitation of morphological processes or dynamical structures. Instead, predefined chemical substrates and their concentration values over time represent the only medium for data storage. Inspired by the idea of successive set-cover completion, a static reaction network to solve a given instance of the exact set cover problem can be constructed. To this end, we generate a pre-compiled exponential workspace comprising up to \( 2^{\vert \mathcal{X} \vert} \) different species. Let \( \mathcal{P}(\mathcal{X}) \) be the power set of \( \mathcal{X} \). Each element of \( \mathcal{P}(\mathcal{X}) \) defines a species within the reaction network. We insert a reaction \( \Phi \rightarrow \Theta \) iff \( \Phi, \Theta \in \mathcal{P}(\mathcal{X}) \) and \( \Phi \subseteq \Theta \) and \( \exists S \in \mathcal{F} : (\Theta = \Phi \cup S) \land (\Phi \cap S = \emptyset) \). After generation of the reaction network, it has to be checked whether there is a reaction path from species \( \emptyset \) to species \( \mathcal{X} \). If so, we obtain the solution “yes”, otherwise “no”.

Figure 2 illustrates the construction of the reaction network along with a simulation study for the toy example of the exact set cover problem introduced in Figure 1(a). It appears that generation of the reaction network in an enumerative manner consumes an exponential number of steps since each species has to be examined whether or not it serves as a substrate. In contrast, subsequent identification of a path from \( \emptyset \) to \( \mathcal{X} \) runs fast in general but we have in mind that each species contributes one equation to a system of ordinary differential equations reflecting the temporal behaviour of the entire reaction network based on an appropriate reaction kinetics.

Our excursus to consideration of a static reaction network without inner structure of involved molecular species is aimed to reveal the need of more elegant algorithmic approaches that make use of dynamical spatial structures.
Fig. 2. Topology of a static reaction network resulting from the problem instance in Figure 1(a) (five elements and three subsets). Instead of dynamical structures, a pre-compiled exponential workspace captures an artificial chemistry to problem solution. Each reaction path from species $\emptyset$ to $abcde$ indicates a problem solution. Its presence can easily be checked by an increasing concentration of species $abcde$. For a corresponding simulation study, we assume mass-action kinetics ($k = 0.1$) and initial substrate concentrations of 0 except for species $\emptyset$ whose initial concentration was set to 10.
Particularly, the pre-compilation of the exponential workspace is worth to be revised by mechanisms operating on the fly.

4 Approach 1: A random search based on membranes with receptor-ligand interactions

4.1 Shaping membranes

In Membrane Computing, some modular approaches to P systems have been proposed [8, 16, 20]. The idea is to create a kind of library of parameterised modules or subroutines that can be reused. Such modules are sets of rules, normally associated to one membrane which executes a specific subtask. Therefore, a module could be also look as a membrane class. We consider a membrane class as the abstraction that encapsulates three components: the membrane structure, its local elements, and the rules that define its behaviour. Local elements are the contents (configuration) and external elements who are “controlled” by the membrane (i.e. that appear only in rules of the membrane). These components give a “shape” to the membrane. The system could be defined in a way that not only configuration of the system changes in time, as normally, but also structure and rules could be dynamic.

Gutiérrez-Naranjo et al. [16] present a general framework to program membrane systems using a brute-force technique, that have been applied to several problems known to be NP-complete. They consider several sequential subtasks (generation, calculation, checking and output stages) to attack combinatorial search problems, and a series of subroutines (sets of rules in a membrane) that vary according to the problem. For example, a detector “macro-rule” gives to a specific membrane the responsibility to detect if a given condition is satisfied. Regarding to the limitation of an exhaustive search [17], we consider here that other strategies, including approximate algorithms could be a more general solution for hard problems.

Different kind of structured objects can be considered in membrane systems: nested (embedded) membranes in cell-like systems, graph-based membranes in tissue-like systems that also could emulate polymers, membranes with receptors, membranes with transduction channels, other membrane interfaces, membrane proteins. Some predefined form of rules and simple strategies have been used in P systems to handle with optimisation problems: label changing, membrane polarization, prioritisation of rules, use of cooperative rules, and others. But perhaps more interesting could be to use transitions that modify the structure of membranes: create, divide, dissolve, merge, and, endocytosis and exocytosis[24] rules.

The recently introduced Polymorphic P systems [3], guided by the principle “rules as data”, constitute an example of how a specific conformation of compartments can generate interesting behaviours. A rule $i$ in such system is defined by a membrane $i$ that contains two membranes: $iL$ and $iR$ (left and right hand
side) and a multiset of objects. The entire system consists of a nesting structure of such membrane-rules, where rules can create other rules, i.e., with the emerging effect of self-modifying programs.

In another work, Nishida [28] also exploits the benefits of nested membranes in a model that solves NP-complete problems, where several optimisation sub-algorithms act in each membrane and send candidate solutions to inner membranes, as a kind of distributed evolutionary algorithm. Although more engineered than biologically plausible, these schemas underlie the common point that membranes arrangement makes possible the information processing and therefore, the functionality of the system.

4.2 A P system model for the Exact Set Cover Problem

In nature, complex molecular mechanisms as for example, signal transduction processes, are behind morphological features [20]. Biological cells communicate with their environment through chemical signaling molecules called ligands. This is possible because cells contain different kinds of receptor molecules, normally embedded on the membrane (cell-surface). A ligand binds to the outer part of a specific receptor in a key-lock fashion, modifying its shape and triggering an internal pathway of signals. Examples of external signal molecules are hormones, toxins, neurotransmitters, and growth and survival factors. They act mainly as substrates, inhibitors or activators. The binding sites of the ligands and their affinity, i.e. the strength of binding with the receptor, are determinant factors in this communication process.

Imagine, nature has to solve the exact set cover problem in vivo. Probably, accordingly on the scale and the localization of the inputs (represented by $X$ and $F$), it will evolve (construct) a (normally robust) machinery whose shape and contents comprises the intrinsic rules that are able to produce the right output. A first simple solution could be using a compartment with active channels, or receptors able to bind with adequate ligands (represented by the subsets in $F$). Fig. 3 shows a possible schema of such a system. We will discuss different possibilities to implement an approximate algorithm to solve the problem using the schema presented above, taken into account the structure of the machinery and the localization of the elements.

As can be seen in Fig. 3, one candidate solution accept all or none of the contents of a specific subset, and then, all others subsets with at least one common element (incompatible subsets) are discarded for this solution. Considering subsets and candidate solutions as membranes, this mechanism is similar as the described for mutual mobile membranes [4] where dual objects in both membranes first bind (“agree”) triggering the endocytosis process.

We adapt this mechanism to a simpler one, as we use the P-Lingua 2.0[12] simulation tool and it does not handle with mobile membranes. A candidate solution in our model consists in a nested membrane formed initially by an outer membrane (labeled 1) that contains information about the subsets $S \in F$ and an inner membrane (labeled $C$) that will contain the possible solution. The role of membrane $C$ is to group the selected compatible subsets. Through ligands
In order to solve the exact set cover for the instance showed in Fig. 1(a), an initial membrane with $|F|$ channels or receptors represents the starting point for generation of candidate solutions. The subsets $A, B, C \in F$ are also membranes who take on the role of ligands. Filled arrows indicate transitions between configurations, when one of such subset-membrane, is arrested in a kind of receptor mediated endocytosis. This receptor triggers the inhibition of other incompatible subsets (with at least one content in common), “closes” the channel, and dissolves the engulfed membrane, liberating its content. If we only consider processes that generate the transitions of configurations illustrated by filled arrows, there is no guarantee to get a solution. Not filled arrows indicate that, as the membrane perceives that it cannot cover the set $X$, a reversion process could be necessary to continue the search. This mechanism can be implemented in several ways. One such “recovery” mechanism could be to return randomly one subset, reconstructing the membrane respect to the corresponding disabled channels. In this case, the elements of the subsets should have been marked in order to regroup them.

$(lS\{i\})$ and receptors $(rS\{i\})$ associated with membranes 1 and $C$, respectively, a candidate solution controls that if one subset $S\{i\}$ is selected for this candidate, then all the incompatible subsets are discarded. When no more subsets could be added to the possible solution, the elements that are successfully covered are counted to give the candidate a fitness. If a candidate solution reaches the optimum (a perfect solution), then a “yes” signal is sent to the environment.

Our basic P system model, for the instance presented in Fig. 1(b), considering 5 candidate solutions, could be defined as follows:

```java
@model<transition>
def main(){
    /* initial configuration */
    /* 5 candidate solutions */
    @mu=[\[\],\[\],\[\],\[\],\[\]]\';
    /* one ligand for each subset, and a "No" symbol */
    @ms(1)=lS\{1\},lS\{2\},lS\{3\},lS\{4\},lS\{5\},lS\{6\},lS\{7\},lS\{8\},lS\{9\},lS\{10\},No;
    /* one receptor for each subset, and a counter molecule */
    @ms(C)=rS\{1\},rS\{2\},rS\{3\},rS\{4\},rS\{5\},rS\{6\},rS\{7\},rS\{8\},rS\{9\},rS\{10\},c(0);
```
/* local variables */
let n = 10; /* number of elements to cover */
let k = 10; /* number of subsets */

/* rules */

/* cooperative rules controlling channels */
[ls{1}[rs{1},rs{5},rs{9}]'C --> [s{1}]'C'1;
[ls{2}[rs{2},rs{6},rs{10}]'C --> [s{2}]'C'1;
[ls{3}[rs{3},rs{7}]'C --> [s{3}]'C'1;
[ls{4}[rs{4},rs{8}]'C --> [s{4}]'C'1;
[ls{5}[rs{5},rs{1}]'C --> [s{5}]'C'1;
[ls{6}[rs{6},rs{2}]'C --> [s{6}]'C'1;
[ls{7}[rs{7},rs{3}]'C --> [s{7}]'C'1;
[ls{8}[rs{8},rs{4}]'C --> [s{8}]'C'1;
[ls{9}[rs{9},rs{1}]'C --> [s{9}]'C'1;
[ls{10}[rs{10},rs{2}]'C --> [s{10}]'C'1;

/* if there remain open channels, then use them */
[ls{i}[rs{i},cm]'C --> [s{i},cm]'C'1:1<=i<=k,1<=m<n;

/* count the elements to evaluate the candidate */
[(s{1})]'C --> s{1}[x{1},x{2},x{3}]'C'1;
[(s{2})]'C --> s{2}[x{4},x{5},x{6}]'C'1;
[(s{3})]'C --> s{3}[x{7},x{8}]'C'1;
[(s{4})]'C --> s{4}[x{9},x{10}]'C'1;
[(s{5})]'C --> s{5}[x{1}]'C'1;
[(s{6})]'C --> s{6}[x{4}]'C'1;
[(s{7})]'C --> s{7}[x{7}]'C'1;
[(s{8})]'C --> s{8}[x{9}]'C'1;
[(s{9})]'C --> s{9}[x{2}]'C'1;
[(s{10})]'C --> s{10}[x{6}]'C'1;
[x{i},cm] --> xc{i},cm+1'C: 1<=i<=n,0<=m<n;

/* if the set is covered then send a positive answer to the environment */
[No][cm]'C --> Yes[cm]'C'1;

4.3 Simulation case study

Simulations are performed using the instance of Fig. 1(b). As usual, rules are applied in a maximal parallel manner and both the membranes and the rules are chosen in a non-deterministic manner. We run the model 50 times per experiment, considering 1, 5 and 10 candidate solutions. Table 1 shows the number of successful runs and the averaged fitness of the candidate solutions when the halting configuration is reached. A single simulation halts when no more rules can be applied or a signal "yes" is received by the skin membrane (in this case, if at least one candidate solution reaches the optimum, that leads the run to be successful).

Using this strategy, the algorithm is able to construct candidate solutions with at least a fitness of 6 (i.e. a solutions whose subsets cover exactly 6 of 10 elements). Due to the inherent parallelism, incrementing the number of candidate

---

5 The complete set of control files for the simulations presented in this paper can be found on the authors permanent website at the FSU Jena (http://www.biosys.uni-jena.de/)
Table 1. Number of successful runs and averaged fitness, steps and time at halting condition for different number of candidate solutions (50 runs per experiment). Fitness (number of covered elements) was first averaged per run according to the number of candidates. A step denotes one transition between one configuration to the next, applying all the rules that could be used at a certain time.

<table>
<thead>
<tr>
<th>Candidates</th>
<th>Fitness</th>
<th>Success</th>
<th>Steps</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.72</td>
<td>2/50</td>
<td>11.56</td>
<td>0.062</td>
</tr>
<tr>
<td>5</td>
<td>7.62</td>
<td>9/50</td>
<td>12.8</td>
<td>0.091</td>
</tr>
<tr>
<td>10</td>
<td>7.72</td>
<td>15/50</td>
<td>13.42</td>
<td>0.152</td>
</tr>
</tbody>
</table>

Approach 2: An evolutionary algorithm implemented in rule-based chemistry

Approach 1 allowed us a quick and easy description of a molecular algorithm that does an effective random search by excluding solution candidates if they are not conform to the constraint of ignoring overlapping solutions. We followed the design principles of using an object centered view, where each of the membranes represents a possible solution candidate object. Each of the objects is following its own heuristic reaction path that can lead it to become a perfect solution to the exact set cover problem. This happens fully concurrently, that is, there is no temporal dependency between different global phases of the algorithm that would have to be controlled from the outside.

In the following section, we will setup a different solution to the exact set cover problem in a totally different environment (SRSim) but while preserving these design principles by using molecule graphs instead of membranes. Extending the first approach further, we add molecular descriptions of molecules that rate, select and reproduce solution candidates, migrating from a molecular implementation of a non-gradient random search algorithm to a heuristic optimisation algorithm. In particular, we present an instance of an asynchronous evolutionary algorithm that is purely implemented in rule-based chemistry [21, 15, 14], similar to the evolutionary algorithm that was described by Banzhaf, 1990 [5] but for the exact set cover problem. The population of individuals from the search space, the calculation of the fitness, the reproduction as well as the selection are realised by structured molecules in the reactor and the limited set of reaction rules amongst them.
Also different membrane computing frameworks, like the Membrane Algorithms that were presented by Nishida in 2005 [28], can be used to implement evolutionary algorithms. But in contrast to these, morphological algorithms are rather using local rules instead of global timing schemes or instead of a global spatial division into areas working under a different scheme. The computation happens according to local rules instead. Nonetheless we could imagine that morphological algorithms might be of use as encapsulated algorithm in the Membrane Algorithms framework, as well.

5.1 Genotype and phenotype

Each possible solution \( C_i \in \mathcal{P}(\mathcal{F}) \) to the exact set cover problem problem is represented by a genotype as well as its consequent phenotype in this approach. These two components are realized by a molecule dimer gen - phen. The genotype molecule gen has exactly one docking site for each possible subset \( S_i \in \mathcal{F} \). “Token molecules” \( T \) can attach to each binding site of gen, representing the inclusion of a certain subset \( S_i \), when attached. The phenotype molecule phen on the other site of the dimer has exactly one docking site for each element \( x_i \in \mathcal{X} \) of the set to be covered. A token \( T \) that is attached to a phen molecule represents a covered element \( x_i \), as displayed in Figure 4(a).

The genotype-phenotype mapping, that is the assignment stating which elements \( \{x_1, ..., x_k\} \) belong to which subset \( S_i \), is realised through artificial transfer factors. These transfer factors follow the example of biological transfer-RNAs (tRNAs) [30], which facilitate the mapping between a triplet of DNA bases to a specific amino acid. In our approach, there is exactly one species of transfer molecules trans for each subset \( S_i \). It is loaded with exactly one subset-token molecule that corresponds to the subset \( S_i \in \mathcal{F} \) and a variable number of element-token molecules, each corresponding to one of the elements \( x_i \in \mathcal{X} \). Some examples for transfer molecules are displayed in Figure 4(b). This design has the advantage, that the problem instance specification is independent from the set of rules and we can even dynamically exchange the problem instance, while the “algorithm” is working in a reactor by feeding a different set of transfer molecules into.

Each subset- and element-token molecule \( T \) is of a given subtype, similar to the codon region of a transfer RNA. Hence we can allow a token to dock only to the binding sites that fit to its subtype. In total, we need a number of \(|\mathcal{X}| + |\mathcal{F}|\) different types of tokens to allow specific binding of tokens to the distinct sites of the genotype and phenotype molecules.

A transfer molecule can dock to any fitting free site of the gen - phen dimer with its attached tokens, when they are of the correct type. If and only if a transfer molecule can dock all of its tokens into the docking sites of the gen - phen solution dimer, the transfer molecule can dissociate from its tokens, leaving them with the solution dimer. If there is already a token blocking one of the necessary sites of the gen - phen dimer, the complete transfer molecule with all of its tokens will dissociate after some time. Thus we assure, that the tokens at the gen side corresponding to the selected subsets \( S_i \) are correctly mapped.
to the tokens on the phen side, corresponding to the elements $x_i$. In this way we cannot attach subsets to a solution that lead to a duplicate selection of an element $x_i$. On the other hand, when all the elements from $\mathcal{X}$ are present in a solution, we have also selected an appropriate set of subsets $\{S_a, \ldots, S_k\}$.

(a) An arbitrary solution candidate represented by a gen-phen dimer with attached subset- and element-tokens.

(b) Exemplary transfer factors. The docking site $a$ is loaded with a subset-token and the sites $e$ are loaded with element-tokens.

Fig. 4. Genotype - Phenotype mapping: molecules representing the genotype gen and the phenotype phen of an individual in the evolution. The mapping (a) between subsets and elements is realised by various transfer factors (b) which are specific to the problem instance displayed in Figure 1(b). For this problem instance, a maximum of three element tokens is sufficient. Each token molecule is marked with the name of the binding sites that it will be able to bind to.

### 5.2 Selection and reproduction with modification

When implementing the system like it was reported in the last section, we obtain an effective random search that will automatically avoid choosing combinations of sets with overlapping elements. Nonetheless, when there is a large number of non-overlapping subset attachment pathways that lead to imperfect solutions, the chances for producing a correct solution to the problem are small. And even when one molecule in the reactor found the perfect solution, it would be hard to identify it among all the other wrong solutions. Consequently we add a selection and reproduction operator here, which will help to cover a larger search space and to amplify good solutions.

Both functions, selection and reproduction, are realized by a single molecule Copier which goes through different internal phases. In the beginning, it attaches to one gen-phen dimer, then to an arbitrary other possible solution dimer as
shown in Figure 5. When it has docked to two possible solution candidates, it “decides” which of them to discard and which to take as a template for reproduction.

This decision is made by a small number of \(|\mathcal{X}|\) reaction rules: for each element \(x_i \in \mathcal{X}\), a rule is defined that is applicable, when there is an element token (e.g. \(T\)-e in Figure 5) that is only present on one of the docked solution instances.

The reaction then marks the solution with the lacking element token as dominated, the other one as template. The more elements are present in one solution but missing in another, the higher the probability for the better solution to be marked as template. Still it is a stochastic process, so it is possible that a good solution \(A\) covering many elements can be marked as dominated in comparison with a competing solution \(B\) owning few elements.

Once one of the solution candidates is marked as dominated, it will reject all its tokens, effectively deleting this solution and remove the dominated marking. Now two alternative events can happen: Either the cleared solution can dissociate from the Copier to recruit new transfer molecules and thus generate a new solution candidate randomly. But until the dissociation happens, new transfer molecules can still dock to the cleared solution candidate. Here, the docked Copier molecule prevents transfer molecules to attach subset tokens to the cleared molecule, if its other template solutions does not show this token. The longer the cleared solutions remains at the Copier, the more exact will the template solution reproduced on the cleared solution. Consequently, by modifying the dissociation rate, we can alter the effective mutation rate in this evolutionary algorithm. Nonetheless, a mutation can only mean to leave out a subset-token in the overwritten molecule, not to induce a new subset-token. But after the new solution candidate dissociates from the Copier, it can indeed be attached to a different subset-token.

**Fig. 5.** The Copier molecule realizes both, the selection and reproduction process. The left solution candidate is fitter than the right one, because it has more element-tokens attached to its phenotype side. As a result, the right candidate will be erased and overwritten by the Copier molecule.
5.3 Differences to standard evolutionary algorithms

In comparison to traditional evolutionary algorithms as they are implemented on sequential computers, our approach is asynchronous to begin with, as described by Banzhaf, 1990 [5] or in embodied evolution scenarios [32]. All the components of the evolution system work continuously in the whole system, without separating the system into discrete generations as it is usually done. The evaluation of the fitness is also different, in that no global ranking is done and that no real valued score is calculated. Instead, the fitness of two solution candidates is compared locally and stochastically. Recombination is not implemented in our approach but might be realised by coupling multiple Copier molecules.

The most striking difference to traditional evolutionary algorithms is probably that we do not switch the “environment” between the execution of the evolutionary algorithm and the evaluation of the fitness function as it usually done, when the fitness evaluation appears as a black box in the algorithm. Here instead, all aspects of the evolutionary algorithm: the mutation operators, the reproduction and the selection are described in the rule-based chemistry language BNGL [6, 21].

5.4 Simulation case study

To test the system with ten subsets, which was presented in Figure 1(b), we implemented the asynchronous, local evolutionary algorithm in rule-based chemistry using the BioNetGen Language (BNGL) [6, 21] and simulated it with SR-Sim, our simulation system for spatial and rule-based chemistries [15]. Some snapshots of a docking transfer factor and the general simulation process are shown in Figure 6.

Though the size 10 of the problem instance might seem very small, nonetheless, a fully enumerative approach would still need to produce $2^{10} = 1024$ different tentative solutions and thus an even larger number of molecules would be necessary to stochastically cover a large fraction of them. Instead, the toy system we investigated comprised 10 gen-phen dimers, three Copier molecules and 30 transfer factors for each subset $S_i \in F$. To test different mutation rates, we varied the dissociation rate $k_{off}$ of gen-phen dimers from the Copier molecules. We considered the three cases of $k_{off} = 10^1, 10^{-4}, 10^{-6}$ and the control case of no selection at all. In addition to the reaction system described before, we added rules to “recycle” used transfer factors. Alternatively, an inflow of loaded transfer factors and an outflow of used trans molecules might be used.

To average the results, we simulated 50 instances for each condition of the system for $1.5 \cdot 10^6$ timesteps. Please note that one timestep is not identical to one generation in the evolutionary algorithms sense in our simulation. Most simulation timesteps will only update particle positions from diffusive movement, some will also incorporate chemical reactions. Also, since the algorithm works asynchronously, we cannot distinguish generations. Instead, we measured the number of dissociations of the gen-phen dimers from the Copier molecules over the whole simulation time to be about 140 for $k_{off} = 10^1$, 110 for $k_{off} = 10^{-4}$
and 10 for $k_{\text{off}} = 10^{-6}$ in average. Since we are not interested in the optimisation performance per generation but per time, the plots in Figure 7 are not normalized in respect to the generations.

In Figure 7(a) we plot the number of perfect solutions present in a simulation averaged over 50 simulation runs with different dissociation rates, showing that a dissociation and mutation rate of $k_{\text{off}} = 10^{-4}$ led to the fastest increase of optimal solutions. As we described in Section 3, the problem instance with ten subsets from Figure 1(b) is still relatively easy to solve, meaning that about three percent of any randomly generated solution will be a perfect solution. Also, while evolution with an intermediate mutation rate (x points) produces the best results here, it is tightly followed by a system with an extremely high mutation rate and thus practically no inheritance (plus points). For more difficult instances of the problem, we expect the evolution with intermediate mutation rates to perform

---

**Fig. 6.** Snapshots of the transfer factors while delivering tokens to a solution candidate (a-e) and a visualization of the SRSim simulation of the entire reactor (f)
(a) Averaged (n=50) number of generated perfect solutions. Error bars indicate the standard error of the mean.

(b) Averaged (n=50) quality of the solutions. Error bars indicate the standard error of the mean.

Fig. 7. Development of the solution candidates in the optimising molecular system over time. When omitting the copier molecules (box points), we obtain a perfect solution in a fraction of the simulations close to the theoretical 2.56%, corresponding to a perfect solution in about every fourth simulation run. With the highest mutation and dissociation rate $k_{off} = 10^1$ instead, we obtain a high percentage of perfect solutions in the system relatively fast. This high mutation rate leads to the situation, that effectively no information is passed from a “parent” molecule to its offspring. A cleared solution candidate will dissociate very fast and generate a new potential solution from scratch (plus points). With a lower dissociation rate $k_{off} = 10^{-4}$, the cleared solution candidates will stay with the copier molecule for some time, inheriting some or all of the template solutions features. This $k_{off}$ rate leads to the fastest convergence in our simulations (x points). The lowest dissociation rate delays the whole process of copying a solution very long and leads to a slow convergence (star points).
better in a more distinguished way. Considering the implicitly defined objective function as the number of covered elements, Figure 7(b) shows a very fast initial rise of the average population fitness, followed by a slower increase in average fitness.

6 Discussion

So far, molecular computing methods were typically aiming at solving difficult computing problems by exploiting the vast amount of molecules in reaction vessels to enumerate all possible solutions in cases of high combinatorial complexity. Hence, the exponential efforts of sequential algorithms in runtime is shifted to an exponential effort in material. So while sequential algorithms could solve larger problem instances “in principle” in many years runtime, molecular computing setups could solve these problems “in principle” when using earth-sized amounts of DNA, protein or other molecular computing substrate [17].

In contrast, we follow the path of heuristics that helps to generate relatively good, though not guaranteed perfect, results for a wide range of problems. Our focus here is not on solving NP-hard problems, but to generate relatively good solutions to non-trivial optimisation tasks while staying in the chemical reaction medium. This brings the advantage of having control structures for bioreactors or other artificial chemical systems that could be implemented directly inside the reaction vessel instead of installing sensors, connecting silicon-based computers and feeding controlling actions back through actuators.

In our evolutionary algorithm example, the propagation of the best solution happens slower than generating many solutions of quite good fitness. Considering a possible application of this algorithm as a system that adapts the configuration of a bioreactor dynamically to the local chemical properties by solving complex computations, we see a high potential in such a system.

Nonetheless it might be quite a hard task to find or engineer molecules or membranes that behave exactly in the way that we designed them for. Here too, it might be an advantage to focus on the object-oriented view of morphological algorithms to distinguish between different aspects of such molecules. Instead of searching molecules that fulfill a hardly comprehensible large table of constraints as in the exponential workspace example, the different aspects and dependencies that are necessary for each molecule type are explicitly mentioned in a membrane system or rule-based algorithm definition.

7 Conclusions

We emphasised the idea of morphological algorithms as effective and elegant tools to formulate strategies for solving problems based on shapes and their variation over time. The major advantage of shapes’ utilisation for applications in membrane computing lies in its inherent feature of dedicated encapsulation of rules and data. A mapping between shape (molecular or membrane structure) and its function reveals a high similarity to the object-oriented paradigm...
of programming which is known to avoid possible side effects of globally defined structures by referential transparency. In our morphological algorithms, appropriate molecular key-lock mechanisms manage the desired encapsulation. We exemplified the exploitation of shapes by introduction of two corresponding algorithms for solution of the exact set cover problem known to be NP-complete. Both algorithms reveal the ability of generating the solution space on the fly by self-organising properties coming from molecular assembly. We combine this feature with heuristics adopted from an asynchronous artificial evolution. If needed, the evolution process can be replaced by a constructive complete enumeration of solution candidates. Interestingly, both introduced morphological algorithms have been closely inspired by concepts found in biology: On the one hand, this is endocytosis for receptor-ligand interactions which allows construction of solution candidates by varying receptors. On the other hand, transfer RNAs can control a molecular assembly enriched by a mechanism of removal of undesired solution candidates whose molecular matter undergoes a recycling and can be reused. Although the artificiality of our algorithms is still away from wetlab implementations, we believe that the potential of morphology can contribute to formulate algorithms elegantly in an object-oriented manner and beyond, it can be useful to make molecular and cellular computations more reliable.

Acknowledgements

We gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF, project no. 0315260A) within the Research Initiative in Systems Biology (FORSYS). Furthermore, the research was supported by the NEUNEU project (248992) sponsored by the European Community within FP7-ICT-2009-4 ICT-4-8.3-FET Proactive 3: Bio-chemistry-based Information Technology (CHEM-IT) program. G. Escuela is supported by Universidad Simón Bolívar (Venezuela) and the German Academic Exchange Service (DAAD) Grant A/08/94489. In particular, we express our gratitude to Peter Kreyßig for helpful discussions.

References


Chemical Analog Computers for Clock
Frequency Control Based on P Modules

Thomas Hinze, Christian Bodenstein, Benedict Schau,
Ines Heiland, and Stefan Schuster
Friedrich Schiller University Jena
School of Biology and Pharmacy, Department of Bioinformatics
Ernst-Abbe-Platz 1–4, D-07743 Jena, Germany
{thomas.hinze,christian.bodenstein,benedict.schau}@uni-jena.de
{heiland.ines,stefan.schu}@uni-jena.de

Abstract. Living organisms comprise astonishing capabilities of information processing for efficient adaptation to environmental changes. Resulting chemical control loops and regulator circuits are expected to exhibit a high functional similarity to technical counterparts subsumed by analog computers. A fascinating example is given by circadian clocks providing an endogenous biological rhythm adapted to the daily variation of sunlight and darkness. Its underlying biochemical principle of operation suggests a general functional scheme corresponding to frequency control using phase-locked loops (PLL). From a systems biology point of view, clock systems can be decomposed into specific modules like low-pass filters, arithmetic signal comparators, and controllable core oscillators. Each of them processes analog chemical signals on the fly. We introduce P modules in order to capture structure, behaviour, and interface of pure chemical analog computer models in terms of building blocks along with two simulation case studies. The first one is focused on chemical analog computer components including a controllable Goodwin-type core oscillator while the second one evolves an entire PLL-based frequency control by means of a pure chemical circadian clock model.

1 Introduction

When considering principles of in-vivo molecular information processing, it turns out that persistent data storage and long-term mechanisms commonly make use of distinguishable spatial structures [25]. Toggling between these structures is then interpreted as an inherent computational process in a discrete manner. Gene regulatory networks constitute a typical example [8]. Here, genetic data encoded by the primary DNA structure act as blueprint of a computer program. Via transcription, translation, and post-translational modification, a specific protein is obtained from the corresponding DNA. The protein’s spatial structure significantly determines its function. In signalling pathways prior to gene regulation, transcription factors become successively activated by various phosphorylations
or by binding to a ligand. Hence, the spatial molecular structure of a transcription factor temporarily stores information, which is processed later on for control of gene expression.

Apparently, discrete mechanisms of information processing benefit from a couple of advantages in comparison to analog ones. Due to the energy demand and reactive specificity needed for toggling between spatial molecular structures, a conformational state is quite robust against slight perturbations. This implies a certain degree of reliability in state maintenance. In addition, the relative small number of possible transitions from an existing state into adjacent ones mostly entails determinism by a highly predictable local relationship between cause and resulting effect. Altogether, it appears that controllable discrete mechanisms seem to be ideal candidates for computing purposes even beyond biological systems [15].

The situation becomes more complicated if we move from the microscopic scale at the level of single biomolecules and atomic structures towards a macroscopic scale coinciding with technical capabilities of signal measuring and sensing on the fly. Typically, variable influences on a system under study coming from its environment, so-called external stimuli, induce more or less analog signals like temperature, pH, pressure, brightness, or local time. In practice, their huge amount of quantification levels leads to rational or real numbers, which represent the signal value at a certain point in time. Especially in the field of control loops there is a need to cope with analog signals in a computational manner.

Originally, control loops had been introduced in engineering to achieve a desired dynamical behaviour of a system like adjusting temperature or local time of a clock [2]. We can distinguish between two different forms of what is meant by “adjustment”. On the one hand, it comprises the elimination of external perturbations. In this application scenario, a predefined reference specifies the desired dynamical behaviour while the influence of external stimuli has to be compensated. Here, variations of external stimuli are usually interpreted to be perturbations, for instance if temperature should be kept constant. On the other hand, a control loop can be constructed in order to adapt its dynamical behaviour to the course of an external stimulus. In this case, the deviance between the control loop’s output and the external stimulus has to be minimised. Imagine a radio controlled clock as an example. Throughout this paper, we will consistently focus on the latter of both scenarios.

Numerous technical attempts succeeded in construction of a plethora of control loops operating mechanically, hydraulically, pneumatically, electrically, electronically, or even chemically [2]. Their simplest scheme has in common a general closed feedback loop composed of four basic elements for real-valued signal processing: plant, sensor, controller, and actuator.

**Plant**: The plant (sometimes also called system) is constituted by one or more physical quantities whose temporal behaviour has been controlled. Its temporal input is given by a tuning signal \( v(t) \), which passes through the system leading to its output \( x(t) = P(v(t)) \). The transfer function \( P \) might include signal weakening, delay, or perturbation.
Sensor: The sensor transforms $x(t)$ into the measured output $y(t) = F(x(t))$ where $F$ acts as transfer function. In some cases, the sensor is dispensable if the plant output can be processed directly holding $y(x) = x(t)$.

Controller: The controller compares $y(t)$ to the external stimulus (reference signal) $w(t)$ and calculates the error signal $e(t) = D(w(t), y(t))$. Subsequently, it provides the control signal $u(t) = C(e(t))$. The underlying transfer function $C$ might include integration or differentiation with respect to $t$.

Actuator: The actuator affects the plant by transforming $u(t)$ into the tuning signal $v(t) = A(u(t))$, which feeds back to the plant.

Signal processing is commonly represented by block diagrams containing characteristic curves or transfer functions like $P$, $F$, $D$, $C$, and $A$ that map input or memorised signals into output signals (cf. Figure 1).

Later, control loops came into the scope of life sciences as part of a cybernetic approach to understand biological systems, now preferably called control systems [12]. They benefit from a strict modularisation that allows a clear decomposition of a complex system into interconnected signal processing units [20, 30].

This paper is inspired by the idea of capturing chemical signal processing on the fly by analog computing using membrane systems. We seize this promising concept by introduction of $P$ modules having in mind that membrane systems can be sufficient tools also beyond pure biological phenomena. For instance, chemical engineering deals with macroscaled reaction-diffusion systems delimited by reactor vessels, membranes, filters, or interceptors. Continuously measured values in conjunction with process parameters and permanent signal evaluation and transformation result in a dynamical control system’s behaviour corresponding to a chemical analog computer. We emphasise this similarity by a collection of $P$ modules for fundamental computational tasks. Starting with reaction systems for arithmetic tasks like addition, non-negative subtraction, multiplication, division, and square root, we construct a chemical model of a low-pass filter. Having this building block at hand, we can compose higher-order functions like chemical integrators and differentiators.

In both spheres, biological and technical systems, oscillatory signals play a major role in order to trigger and control time-dependent processes [1, 2]. Biochemical core oscillators are simple devices for the generation of continuously running clock signals by periodically alternating substrate concentrations [17, 29]. To this end, signal processing units can suffice, which consist of astonishingly simple reaction networks comprising at least one feedback loop. So, it is no surprise that probably numerous evolutionary origins led to oscillatory reaction
networks, while independently technical attempts succeeded in construction of single clocks or clock generators.

Exploration of chronobiological systems emerges as a growing research field within bioinformatics focusing on various applications in medicine, agriculture, and material sciences [5]. From a systems biology perspective, the question arises whether biological control systems for regulation of oscillatory signals and their technical counterparts utilise similar mechanisms. If so, modelling approaches and parameterisation adopted from a strict modularisation can help to identify general components for frequency control in circadian clock systems along with gaining comprehensive insight into mechanisms of clock maintenance, synchronisation, and entrainment to external stimuli like the daily rhythm of sunlight and darkness.

Circadian rhythms embody an interesting biological phenomenon that can be seen as a widespread property of life. The coordination of biological activities into daily cycles provides an important advantage for the fitness of diverse organisms [28]. Based on self-sustained biochemical oscillations, circadian clocks are characterised by a period of approximately 24h that persists under constant conditions (like constant darkness or constant light). Their ability for compensation of temperature variations in the physiological range enables them to maintain the period in case of environmental changes. Furthermore, circadian clocks can be entrained. This property allows a gradual reset of the underlying oscillatory system for adjustment by exposure to external stimuli like daily variations of brightness or daytime–nighttime temperature cycles.

Circadian clock systems appear to be special forms of frequency control systems. Following this line, it should be possible to identify appropriate interacting modules representing elements of a dedicated control-loop model. Indeed, coupling of a controllable core oscillator with a low-pass filter and a multiplicator suffices to reproduce the desired entrainment behaviour of a circadian clock. We introduce the according system composed of P modules along with simulation results. To our best knowledge, this is the first model of a pure chemical frequency control system based on a phase-locked loop (PLL, [3]).

We believe that the concept of P modules can contribute to bring together the area of membrane computing with systems theory [24] and analysis towards applications in control of continuous or sampled signals. For prospective lines of research, membrane systems constitute a powerful tool to cope with signals encoded by dynamical structures instead of real values.

2 P Modules

The concept of P modules is based on the assumption that ‘structure follows function’. This implies constraints and limitations in the variety of valid network topologies. Nevertheless, there is a plethora of different strategies and implementations to achieve a certain complex network function. The pool of sufficient network candidates can be divided into compositions of a low number of elementary functional units called modules. This term is not new in systems
biology when considering recurrent motifs conserved in metabolic, cell signalling, and gene regulatory networks [6, 14]. We extend the notion of modules in terms of information processing: In this context, a P module is able to fulfill an elementary computational task. When combining those modules towards reaction networks capable of a more complex functionality, we permit shared molecular species among distinct modules. This way of module coupling enables compact network topologies in accordance with evident observations from in-vivo studies [23]. Moreover, there is no need for further separate specifications of extramodular channels. In most cases, the behaviour of a module can be captured by chemical counterparts of transfer functions in conjunction with characteristic curves, which exhibit an established practice in engineering. Utilisation of transfer functions for modules significantly reduces the number of distinct parameters to be considered by keeping the relevant characteristics of the entire network.

A P module is a construct to specify the interface of a generally real-valued system able to process temporal signals. Particularly, the temporal course of non-negative substrate concentrations or physical conditions (like Kelvin temperature, inner and outer pressure, or brightness detected by a photo cascade) are signals managed by a P module.

Each P module represents a container encapsulating an explicite specification of its dynamical behaviour. A collection of prototypic specification examples includes:

- metabolic P systems, for instance of the form
  \( M = (X, R, V, H, \Phi, \nu, \mu, \tau) \), cf. [4, 10, 21]

- P systems for cell signalling modules (CSM) of the form
  \( \Pi_{\text{CSM}} = (V, V', R_1, \ldots, R_r, f_1, \ldots, f_r, A, C, \Delta \tau) \), cf. [17]

- P systems for cell signalling networks (CSN) of the form
  \( \Pi_{\text{CSN}} = (V, V', E, M, n) \), cf. [16]

- ordinary differential equations (ODEs) in conjunction with an appropriate numerical solver. The ODEs should be derived from reaction or diffusion kinetics, cf. [7].

- a transfer function on its own, either given explicitly or implicitly by a mathematical term or alternatively by a table of numeric values (characteristic curve) along with an algorithm for interpolation, approximation, or regression, cf. [2]

We define a P module by a triple

\[ (\downarrow, \uparrow, \square) \]

where \( \downarrow = (I_1, \ldots, I_i) \) indicates a finite enumerative list of input signal identifiers, \( \uparrow = (O_1, \ldots, O_o) \) a finite enumerative list of output signal identifiers, and \( \square \) the underlying system specification processing the input signals and producing the output signals with or without usage of auxiliary inherent signals not mentioned in the P module’s input-output interface. Each signal is assumed to represent a real-valued temporal course, hence a specific function \( \sigma : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R} \) (\( \mathbb{R}_{\geq 0} \): non-negative real numbers).
Attention must be paid to the composition of P modules to keep signal identifiers as well as signal semantics consistent when migrating from one P module to another. To this end, we allow for a signal-specific export-import mapping, which expresses the equivalence of identifiers (shared molecular species) as well as unification of signal values. Unification of signal values might include conversion between units of measurement, for instance translation between molar abundance and substrate concentration, which exploits the spatial capacity of the reaction vessel as an internal parameter of the corresponding P module.

Temporal simulation of a system’s behaviour resulting from composition of several P modules requires a global clock. In this context, we demand that unification of signal values across P modules does not produce any time delay concerning the assumed global clock.

3 Chemical Analog Computer Components

3.1 Mass-action Kinetics

For description of the temporal behaviour of chemical reaction networks we consider substrate concentrations over time presuming homogeneity in reaction space. General mass-action kinetics \[7\] formulates reaction system’s dynamics subject to production and consumption rates \(v_p\) and \(v_c\) of each substrate \(S\) in order to change its concentration by 

\[
\frac{d[S]}{dt} = v_p - v_c.
\]

A reaction system with a total number of \(n\) substrates and \(r\) reactions 

\[
a_{1,1}S_1 + a_{2,1}S_2 + \ldots + a_{n,1}S_n \xrightarrow{k_1} b_{1,1}S_1 + b_{2,1}S_2 + \ldots + b_{n,1}S_n
\]

\[
a_{1,2}S_1 + a_{2,2}S_2 + \ldots + a_{n,2}S_n \xrightarrow{k_2} b_{1,2}S_1 + b_{2,2}S_2 + \ldots + b_{n,2}S_n
\]

\[
a_{1,r}S_1 + a_{2,r}S_2 + \ldots + a_{n,r}S_n \xrightarrow{k_r} b_{1,r}S_1 + b_{2,r}S_2 + \ldots + b_{n,r}S_n
\]

employs stoichiometric factors \(a_{i,j} \in \mathbb{N}\) (reactants), \(b_{i,j} \in \mathbb{N}\) (products) and kinetic constants \(k_j \in \mathbb{R}_{>0}\) assigned to each reaction quantifying its velocity (\(\mathbb{N}\): natural numbers, \(\mathbb{R}_{>0}\): positive real numbers). The corresponding ODEs read \[9\]:

\[
[S_i] = \frac{d}[S_i]}{dt} = \sum_{h=1}^{r} \left( k_h \cdot (b_{i,h} - a_{i,h}) \cdot \prod_{l=1}^{n} [S_l]^{a_{l,h}} \right) \quad \text{with} \quad i = 1, \ldots, n
\]

In order to obtain a concrete trajectory, all initial concentrations \([S_i](0) \in \mathbb{R}_{\geq 0}\), \(i = 1, \ldots, n\) are allowed to be set according to the relevance for the reaction system.

We adopt mass-action kinetics for all chemical analog computer components introduced within this section: addition, non-negative subtraction, multiplication, division, square root, and low-pass filter (also delay cascade) followed by a sketch of integrators and differentiators. More complicated mathematical expressions can be composed by elementary functions (e.g. by means of Taylor
series). Initial concentrations of dedicated substrates $X_i$ act as input while the concentration of the output substrate $Y$ provides the result typically within its steady state. In this context, the steady state is the only stable fixed point reached asymptotically from valid initial substrate concentrations. According to the assumed kinetic constants $k_j$, there is mostly a latency before approaching an appropriate $\varepsilon$-neighbourhood of the steady state (except in rare cases without any change of the output substrate) with $\varepsilon(t) = \lim_{t \to \infty} |S_i(t) - [S_i](t)|$.

The chemical analog computer components described below have been intended to be “minimalistic”. The plausibility of minimalism is emphasised by the fact that the SBML evolver [19], a software tool for enumeration and fitness measure of reaction networks undergoing mass-action kinetics, could not obtain smaller networks (although this is not a proof in a mathematical sense). Due to the aim of minimalism, we do not require mass conservation within the reaction networks. If needed, mass conservation can be achieved by addition of side reactions incorporating balanced supplier and waste substrates.

### 3.2 Addition

The $P$ module $\Pi_{\text{add}} = ((X_1, X_2), (Y), M)$ carries out the addition of both non-negative input substrate concentrations $[X_1]$ and $[X_2]$ producing the output concentration $[Y]$ by the ODEs $M$:

$$
\begin{align*}
\dot{[X_1]} &= 0 \\
\dot{[X_2]} &= 0 \\
\dot{[Y]} &= k_1[X_1] + k_2[X_2] - k_3[Y]
\end{align*}
$$

Let $k_1 = k_2 = k_3 > 0$. The functional principle is a direct consequence of the summation effect on substrate production caused by separate reactions contributing summand-like portions. We assume the regeneration of consumed input substrates by an autocatalytic loop. Figure 2 illustrates the module topology.

Fig. 2. “3 + 2 = 5”; parameter setting: $k_1 = k_2 = k_3 = 0.1, [X_1](0) = 3, [X_2](0) = 2, [Y](0) = 0$
The transfer function is given by $[Y] = [X_1] + [X_2]$.

### 3.3 Non-negative Subtraction

The P module $H_{\text{nsub}} = ((X_1, X_2), (Y), M)$ performs the non-negative subtraction $[X_1] - [X_2]$ producing the output concentration $[Y]$ by the ODEs $M$:

$$
\begin{align*}
\dot{X}_1 &= 0 \\
\dot{X}_2 &= 0 \\
\dot{Y} &= -k_2[Y][Z] - k_1[Y] + k_1[X_1] \\
\dot{Z} &= k_1[X_2] - k_2[Y][Z]
\end{align*}
$$

Let $k_1 > 0$ and $k_2 > 0$. The network operates in a way to transfer the amount of $X_1$ to $Y$ while simultaneously the amount of $X_2$ becomes removed from $Y$ by a decay via $Z$. In case of $[X_1] \geq [X_2]$, the final concentration $[Y] = [X_1] - [X_2]$ remains. The presence of $Z$ ensures an output of zero iff $[X_1] < [X_2]$. We assume the regeneration of consumed input substrates by an autocatalytic loop. Figure 3 illustrates the module topology along with an example. The system reaches its steady state comprising

$$
[Y](\infty) = \begin{cases} 
[X_1](0) - [X_2](0) & \text{iff } [X_1](0) > [X_2](0) \\
0 & \text{otherwise}
\end{cases}
$$

Estimation of the steady state can be done using the stationary fixed-point condition based on the eigenvalues of the Jacobian matrix of $M$. For $[X_1](t) > [X_2](t)$,
all eigenvalues are \(<0\), which implies that the steady state is asymptotically stable. In case of \([X_2](t) > [X_1](t)\), we reach an unstable fixed point maintained by the permanent outflow via \(X_2\) and \(Z\). \([X_1](t) = [X_2](t)\) results in no finite fixed point since the steady state value for \(Z\) diverges while \(Y\) converges to zero. The transfer function is given by 

\[
[Y] = \begin{cases} 
[X_1] - [X_2] & \text{iff } [X_1] > [X_2] \\
0 & \text{otherwise}
\end{cases}
\]

### 3.4 Multiplication

The P module \(\Pi_{\text{mul}} = ((X_1, X_2), (Y), M)\) executes the multiplication according to the transfer function 

\[
[Y] = [X_1] \cdot [X_2]
\]

by the ODEs \(M\):

\[
\begin{align*}
\dot{[X_1]} &= 0 \\
\dot{[X_2]} &= 0 \\
\dot{[Y]} &= k_1 [X_1] [X_2] - k_2 [Y]
\end{align*}
\]

Let \(k_1 = k_2 > 0\). A dimerisation suffices to emulate an arithmetic multiplication using mass-action kinetics. We assume the regeneration of consumed input substrates by an autocatalytic loop. Figure 4 illustrates the module topology along with an example. The system reaches its steady state comprising 

\[
[Y](\infty) = \lim_{t \to \infty} (1 - e^{-k_1 t}) \cdot ([X_1](t) \cdot [X_2](t)) = [X_1](0) \cdot [X_2](0).
\]

**Fig. 4.** “3·2 = 6”; parameter setting: \(k_1 = k_2 = 0.1, [X_1](0) = 3, [X_2](0) = 2, [Y](0) = 0\)

### 3.5 Division

The P module \(\Pi_{\text{div}} = ((X_1, X_2), (Y), M)\) divides \([X_2]/[X_1]\) producing the output concentration \([Y]\) by the ODEs \(M\):

\[
\begin{align*}
\dot{[X_1]} &= 0 \\
\dot{[X_2]} &= 0 \\
\dot{[Y]} &= k_2 [X_2] - k_1 [X_1] [Y]
\end{align*}
\]
Let \( k_1 = k_2 > 0 \). This network employs the divisor \( X_1 \) as a catalyst for the decay of \( Y \) while \( Y \) is supplied by \( X_2 \). We assume the regeneration of consumed input substrates by an autocatalytic loop. Figure 5 illustrates the module topology along with an example. The system reaches its steady state comprising

\[
[Y](\infty) = \begin{cases} 
  \lim_{t \to \infty} \left( 1 - e^{-k_1 t} \right) \cdot \frac{[X_2](t)}{[X_1](t)} & \text{iff } [X_1](t) > 0 \\
  \lim_{t \to \infty} \left( \int k_2 [X_2](t) \, dt \right) & \text{otherwise}
\end{cases}
\]

![Diagram of network](image)

Fig. 5. “6/3=2”; parameter setting: \( k_1 = k_2 = 0.1, [X_1](0) = 3, [X_2](0) = 6, [Y](0) = 0 \)

The transfer function is given by

\[
[Y] = \begin{cases} 
  \frac{[X_2]}{[X_1]} & \text{iff } [X_1] > 0 \\
  \to \infty & \text{iff } [X_1] = 0 \text{ and } [X_2] > 0 \\
  0 & \text{iff } [X_1] = 0 \text{ and } [X_2] = 0
\end{cases}
\]

### 3.6 Square Root

The \( P \) module \( \Pi_{\text{sqrt}} = ((X), (Y), M) \) calculates the transfer function \( Y = \sqrt{X} \) by the ODEs \( M \):

\[
\begin{align*}
\dot{X} &= 0 \\
\dot{Y} &= k_1 [X] - 2k_2 [Y]^2
\end{align*}
\]

The weighted balance between the stoichiometrically doubled decay of \( Y \) in comparison to its production from \( X \) induces the effect of square-root estimation. We assume the regeneration of consumed input substrates by an autocatalytic loop. Figure 6 illustrates the module topology along with an example. Let \( k_1 = 2k_2 > 0 \). Under this condition, \( M \) implies the solution

\[
[Y](t) = \sqrt{[X](t)} \cdot \tanh(k_1 t \sqrt{[X](t)})
\]

For \( [X](t) = [X](0) \) and \( t \to \infty \), we obtain

\[
[Y](\infty) = \sqrt{[X](0)}
\]
Fig. 6. \(\sqrt{2} = 1.414\ldots\); parameter setting: \(k_1 = 0.2, k_2 = 0.1, [X](0) = 2, [Y](0) = 0\)

### 3.7 Low-pass Filter

Acting as a moving average element, a simple variant of a chemical low-pass filter can be constructed by a linear reaction cascade. The sequence of consecutively running reactions causes a successive delay along with smoothing of the passing signals. Each cascade stage buffers and accumulates the arriving molecules for a varying time span before transduction to the next stage. This time-limited accumulation is mainly responsible for the filtering effect on oscillatory input signals. While low frequency signals pass the filter, higher frequency oscillations become more and more diminished and hence eliminated. In addition, oscillatory waveforms undergo a conversion into a sinusoidal shape since higher-order harmonics get lost. The behaviour of a low-pass filter can be specified by a so-called Bode plot, which depicts the intensity of signal weakening subject to different frequencies. Interestingly, biochemical low- and band-pass filters had already been identified in signalling cascades of intracellular processes [22, 27].

Assuming a low-pass filter composed of \(n\) stages, an according P module \(\Pi_{lpf} = ((X), (Y), M)\) is specified by the ODEs \(M\):

\[
\begin{align*}
\dot{[X_1]} &= k_1[X] - k_2[X_1] \\
\dot{[X_2]} &= k_2[X_1] - k_3[X_2] \\
\dot{[X_3]} &= k_3[X_2] - k_4[X_3] \\
& \vdots \\
\dot{[X_{n-1}]} &= k_{n-1}[X_{n-2}] - k_n[X_{n-1}] \\
\dot{[Y]} &= k_n[X_{n-1}] - k_{n+1}[Y]
\end{align*}
\]

The transfer function for non-oscillatory input signals is described by a smoothing delay. For oscillatory input signals apart from an analytical evaluation, the transfer function becomes replaced by a characteristic curve, which arises from simulation studies or measure. In some cases, preferably for sinusoidal signals and those entirely captured by a finite Fourier series, the temporal behaviour can be analytically mapped into a frequency domain using Laplace transform. A Bode plot describes the characteristic curve of a low-pass filter by...
Fig. 7. Simulation of a 5-stage chemical low-pass filter affecting a polyfrequential input signal given by substrate concentration course \([X](t)\). This input course was obtained by multiplication of two arbitrarily chosen sinusoidal signals \(f(t)\) and \(g(t)\) with periods of one day and two days: \(f(t) = 7 + 6.9 \sin \left( \frac{2\pi}{1} \cdot t \right)\), \(g(t) = 7 + 6.9 \sin \left( \frac{2\pi}{2} \cdot t \right)\) resulting in \([X](t) = f(t) \cdot g(t) = 49 + 48.3 \sin \left( \frac{2\pi}{1} \cdot t \right) + 48.3 \sin \left( \frac{2\pi}{2} \cdot t \right) + 23.805 \cos \left( \frac{2\pi}{1} \cdot t \right) + 23.805 \cos \left( \frac{2\pi}{2} \cdot t \right)\). The output course \([Y](t)\) reveals the filtering effect by providing a monofrequential signal of almost sinusoidal shape with period length of two days after transient phase. The higher frequency signal with period length of one day became eliminated; parameter setting: \(k_1 = k_2 = k_3 = k_4 = 0.036, k_5 = 3600, k_6 = 180, [X_i](0) = 0\) for \(i = 1, \ldots, 4, [Y](0) = 0\).

Fig. 8. Bode plot derived from the low-pass filter in Figure 7. Slope: \(\approx -100\) dB per frequency magnitude; cutoff frequency: \(\approx 1.04 \cdot 10^{-5}\) s\(^{-1}\). This value coincides with a period length of approx. 1.13 days.
pointing out two essential parameters, the cutoff frequency and the slope, see Figure 8. The cutoff frequency marks the transition from the passband to the stopband whereas the signal amplitude becomes more and more weakened. Usually, the relative intensity of weakening is given logarithmically in magnitudes \( \text{dB} = 10 \cdot \lg \left( \frac{\text{amplitude of output signal}}{\text{amplitude of input signal}} \right) \). The slope characterises the increase of damping within the stopband. Once estimated, either parameters – cutoff frequency and slope – are enough from a systemic point of view to determine the behaviour of a low-pass filter instead of managing numerous kinetic constants and initial concentrations.

Figure 7 exemplarily depicts the effect of a 5-stage low-pass filter to a polyfrequent input concentration course \([X](t)\) while Figure 8 shows the corresponding Bode plot. The number of stages \(n\) define the degree of asymptotic smoothing into a sinusoidal output concentration course after transient phase.

### 3.8 Integrator and Differentiator

By combining a low-pass filter with a P module for addition, we can construct a simple integrator module inspired by the idea of summation of \(l\)-latest sampled signals:

\[
\int_{a}^{b} f(t) \, dt \approx \sum_{i=0}^{l-1} f(t_i) \quad \text{with} \quad t_i = a + i \cdot \frac{b-a}{l} \quad \text{and} \quad l \in \mathbb{N} \setminus \{0\}
\]

Figure 9 illustrates a schematic diagram of a chemical integrator. The filter cascade on top induces successive delays while the input signal passes through. The adder accumulates the resulting signal courses towards a numerically sampled integration. Due to the final number \(l\) of sampling intervals, the integrator runs over into saturation.

A differentiator can be built following a similar idea. Having in mind that

\[
\frac{df}{dt} \approx \frac{f(t + \Delta t) - f(t)}{\Delta t}
\]

for a sampling interval \(\Delta t\), the combination of P modules for non-negative subtraction with a low-pass filter suffices. Since the slope of \(f\) can be either positive or negative, it makes sense to plan two subtraction modules. One of them exclusively detects positive slopes \((f(t + \Delta t) > f(t))\), while the other one indicates negative slopes, respectively.

![Schematic diagram of a simple 3-stage chemical integrator module](image)

**Fig. 9.** Schematic diagram of a simple 3-stage chemical integrator module; parameter setting: \(k_i = 0.1\) for \(i = 1, \ldots, 6\), \([X](0) = 5\), \([X_1](0) = 0\), \([X_2](0) = 0\), \([Y](0) = 0\).
4 Clock Frequency Control using Phase-locked Loops

A case study of our modularisation concept addresses the biological frequency control of circadian clocks. From a systems point of view, they constitute biochemical regulatory circuits whose functionality resembles technical counterparts utilising phase-locked loops [3]. Corresponding circuits comprise three modules:

1. a core oscillator (plant) whose frequency has been controlled to adapt to an external stimulus. The intensity and periodicity of environmental light converted into a specific protein abundance by a photo cascade represents a typical external stimulus.

2. a signal comparator (controller), for instance a phase detector, responsible for determining the deviation between the signal produced by the core oscillator on the one hand and the external stimulus on the other. The signal comparator carries out an arithmetic task.

3. a biochemical low-pass filter (actuator) completes the control system by providing a global feedback loop.

4.1 A Controllable Goodwin-type Core Oscillator

There are numerous biochemical core oscillators found in living organisms’ circadian clocks. From today’s perspective, the majority of core oscillators reveals the Goodwin type [11] by involving at least one inhibiting effect on gene expression. Furthermore, core oscillators can be of post-translational type exploiting a cyclic scheme of protein phosphorylation and dephosphorylation in conjunction with complex formation and decomposition [23]. Each core oscillator must be able to vary its frequency according to the tuning signal produced by the low pass filter.

Let us first consider a controllable Goodwin-type core oscillator on its own. A Goodwin-type oscillator comprises an abstract model of a cyclic gene regulatory network, which is able to exhibit a sustained oscillatory behaviour in its substrate concentrations. Goodwin-type oscillators have in common three dedicated substrates typically called $X$, $Y$, and $Z$ in which $X$ represents a mRNA translated into a protein $Y$ within the cytoplasm. $Y$ is transported into the nucleus where it functions as a repressor $Z$, which in turn inhibits the transcription of $X$. All components ($X$, $Y$, and $Z$) degrade in the presence of specific proteases acting as catalysts. It turns out that the velocity of degradation is the most effective way to control the oscillation frequency [26].

The original Goodwin oscillator, a prototypic core model for generation of circadian rhythms, comes with three attributes worth to be revised towards a more biochemical model [13]: First, the inhibition of $X$ utilises a Hill term (saturation kinetics, [7]) whose Hill coefficient demands an unrealistically high value of 9 or higher to ensure sustained oscillations. Since the Hill coefficient typically coincides with the number of reactive binding sites assigned to the repressor protein $Z$, one would normally expect values below or equal to 4. Second, the intracellular transportation of $Y$ into the nucleus requires an additional transportation term ($-k_5[Y]$, cf. Equation 1) that is not included in the original
model. Finally, we assume saturation for the degradation of X, Y, and Z due to two different possible reasons: One is a limited enzymatic capability and the second, more likely one is the tight regulation of protein degradation. We will show in the following section how the latter can influence circadian regulation.

To overcome the insufficiencies of the original model, we employ a slightly modified version, which utilises Michaelis-Menten kinetics [7] instead of mass-action kinetics for degradation and additionally we take into account the transportation of Y. Interestingly, the resulting model reaches sustained oscillations with lower Hill coefficients $h$ in Hill terms of the form $\frac{a}{\beta + \gamma S^n}$ (values $\geq 1$ suffice here, we choose 2). The resulting ODEs read:

$$\frac{dX}{dt} = \frac{a}{A + K_1[Z]^2} - \frac{k_2[X]}{K_2 + [X]}$$
\[ \dot{Y} = k_3X - k_5Y - \frac{k_4Y}{K_4 + [Y]} \]  
\[ \dot{Z} = k_5Y - \frac{k_6Z}{K_6 + [Z]} \]

Figure 10 reveals the behaviour of our Goodwin-type core oscillator with respect to its incorporation into a PLL-based frequency control loop. The upper-right diagram gives a notion of the waveforms generated by the oscillator, which become visible after a short transient phase of approx. 1.5 days. It turns out that \( X \) and \( Y \) demonstrate spike-like pulses emerging from a ground plateau (also found in the original Goodwin model). In contrast, the course of \( Z \) resembles a more sinusoidal shape. This advantageous feature makes the oscillatory signal of \( Z \) easier than the others to compare with an oscillatory external stimulus.

Hence, we decide to consider the temporal concentration course of \( Z \) as output signal. Supplementary diagrams within Figure 10 show the almost linear dependency of the oscillator’s period length on the velocity parameters \( k_2 \), \( k_4 \), and \( k_6 \) of the corresponding degradation reactions. Therefore, regulated degradation by proteases affecting one or more of these velocity parameters can control the oscillatory frequency, which is the reciprocal of the period length.

4.2 Goodwin-type Circadian Clock Control by Phase-locked Loop

We construct a PLL-based chemical model of a frequency control system by composition of a Goodwin-type core oscillator (plant) with an arithmetic signal comparator (controller) and a low-pass filter (actuator). For the resulting entire system topology, see Figure 11.

The signal comparator is responsible for estimation of the phase shift between the oscillatory signal \( [Z](t) \) released by the core oscillator and the external stimulus \( [E](t) \) as reference signal. Interestingly, a simple arithmetic signal multiplication is sufficient if both oscillatory signals exhibit a sinusoidal or almost sinusoidal course, let us assume \( \sin(\omega_1 t + \phi) \) and \( \cos(\omega_2 t + \psi) \). Multiplication leads to \( \sin(\omega_1 t) \cdot \cos(\omega_2 t + \psi) = \frac{1}{2} (\sin((\omega_1 - \omega_2)t - \psi) + \sin((\omega_1 + \omega_2)t + \phi)) \) due to elementary trigonometric laws. While the term \( \sin((\omega_1 + \omega_2)t + \phi) \) exclusively comprises the frequency sum \( \omega_1 + \omega_2 \), it becomes eliminated by passing the low-pass filter and hence it can be neglected. The remaining term \( \sin((\omega_1 - \omega_2)t - \psi) \) indicates the estimated phase shift to be minimised. To do so, the external stimulus’ phase becomes locked while the core oscillator has been forced to “catch it up” by temporarily enhancing its frequency until the phase shift is compensated. Our signal comparator utilises the P module \( \Pi_{\text{mul}} \).

The low-pass filter \( \Pi_{\text{lpf}} \) desensitises the control system’s global feedback loop. In order to emulate the behaviour of a circadian clockwork, the cutoff frequency should reflect a desired period length of almost one day. We decided to employ a 5-stage low-pass filter as described in the previous section whose kinetic constants had been increased in order to accelerate the corresponding reactions towards a slightly higher cutoff frequency.
Fig. 11. Chemical model of a PLL-based frequency control system comprising modules of a controllable core oscillator, a signal comparator, and a low-pass filter specified by underlying reaction kinetics or alternatively by corresponding transfer functions or characteristic curves. Modules are interconnected via shared molecular species.

We plan to affect the core oscillator’s frequency by control of $Z$ degradation velocity. This requires an appropriate mapping from the low-pass filter’s output signal $[F](t)$ (tuning signal) into a substrate concentration assumed to act catalytically in terms of a protease $D$. In addition, we must pay attention to ensure that all involved substrate concentrations remain non-negatively. To this end, it is recommended to scale the tuning signal with a constant scaling factor $0 < a_e < 1$ using $[D](t) = 1 + a_e \cdot [F](t)$, which allows an almost linear variation of $Z$ degradation velocity [18]. We implement the scaling by combination of corresponding arithmetic modules. The resulting core oscillator had been captured by the P module $\Pi_{osc}$.

The entire system along with an appropriate parameter setting appears in P-modular form where $[F^{**}]$, $[X^{**}]$, and $[Y^{**}]$ denote concentrations of shared molecular species leaving or entering a module:

core oscillator $\Pi_{osc} = ((F), (Z), ODE_{osc})$ with

\[
[U] = 1 \text{ (const)}
\]

\[
[D] = k_7[U] - k_7[D] + k_8[F], \quad [D](0) = 0
\]

\[
[X] = \frac{\alpha}{A + K_1[Z]^2} - \frac{k_2[X]}{K_2 + [X]}, \quad [X](0) = 0
\]

\[
[Y] = k_3[X] - k_5[Y] - \frac{k_4[Y]}{K_4 + [Y]}, \quad [Y](0) = 0
\]
\[
\dot{Z} = k_5 [Y] - \frac{k_6 [D] [Z]}{K_6 + [Z]}, \quad [Z](0) = 5
\]
\[
[F^{**}] = -k_a [F]
\]

\[
a = 6, A = 0.6, K_1 = 1, K_2 = 0.2, k_2 = 3.4, k_3 = 0.3
\]
\[
K_4 = 0.2, k_4 = 2.2, k_5 = 0.1, K_6 = 1.44, k_6 = 1.3
\]
\[
k_7 = 1000, k_8 = 100
\]

signal comparator \( \Pi_{mul} \) = \((E, Z), (Y), \text{ODE}_{mul}\) with
\[
[\dot{Y}] = m_1 [E] [Z] + [Y^{**}], \quad [Y](0) = 0, \quad m_1 = 100
\]

low-pass filter \( \Pi_{lpf} \) = \((X), (F), \text{ODE}_{lpf}\) with
\[
[\dot{X}_1] = l_1 [X] - l_2 [X_1], \quad [X_1](0) = 0
\]
\[
[\dot{X}_2] = l_2 [X_1] - l_3 [X_2], \quad [X_2](0) = 0
\]
\[
[\dot{X}_3] = l_3 [X_2] - l_4 [X_3], \quad [X_3](0) = 0
\]
\[
[\dot{X}_4] = l_4 [X_3] - l_5 [X_4], \quad [X_4](0) = 0
\]
\[
[\dot{F}] = l_5 [X_4] + [F^{**}], \quad [F](0) = 0
\]
\[
[X^{**}] = -l_1 [X]
\]
\[
l_1 = l_2 = l_3 = 0.108, l_4 = 3600, l_5 = 180
\]

shared molecular species (intermodular equivalence of signal identifiers)

\( \Pi_{mul} \rightarrow \Pi_{lpf} \) : \( X = Y \) (signal transduction from \( \Pi_{mul} \) to \( \Pi_{lpf} \))
\( \Pi_{lpf} \rightarrow \Pi_{osc} \) : \( F = F \) (signal transduction from \( \Pi_{lpf} \) to \( \Pi_{osc} \))
\( \Pi_{osc} \rightarrow \Pi_{mul} \) : \( Z = Z \) (signal transduction from \( \Pi_{osc} \) to \( \Pi_{mul} \))

imported external signals \( [E] \rightarrow \Pi_{mul} \)

The chosen parameter setting of the low-pass filter entails a cutoff frequency of approx. \( 1.9 \cdot 10^{-5} \text{s}^{-1} \), which equals to a period length of approx. 0.61 days in accordance with the need for a circadian clock system. We intend to run the arithmetic operations (addition and multiplication) fast to shorten its latency. That’s why it makes sense to set comparatively high values for the relevant kinetic constants.

Having a suitably parameterised system at hand, we conducted a couple of simulation studies to explore whether the overall behaviour of our control system model coincides with expectations for circadian clocks.

In a first scenario, the external stimulus persists at a constant value \([E]\) according to permanent darkness or permanent light. We expect a so-called free-running sustained oscillation of \([Z]\) whose period might be influenced by the constant external stimulus. Figure 12A confirms this dependency by an almost linear relation. The period length becomes shortened by ascending values of the external stimulus while permanent darkness (\([E](t) = 0]\) indicates the maximum period length (approx. 25h30’ using the abovementioned parameter setting).
Fig. 12. Simulation scenarios exploring the behaviour of our PLL-based control system model. A: Period lengths of sustained oscillations subject to constant external stimulus, B: Time to entrainment to external stimulus whose frequency initially differs from those of core oscillator, C: Ability of our control system to cope with both, different frequency and different initial phase between external stimulus and core oscillator, D: Comparison of best case and worst case from entrainment run to 24h subsumed in C.

A second simulation study is inspired by gaining insight into the time to entrainment if \([E]\) and \([Z]\) start to oscillate with no phase shift but comprising distinct initial frequencies instead. We choose a sinusoidal course of the external stimulus \([E]\). The core oscillator is configured to exhibit a natural period of \(24\text{h}12'\) while the period of the external stimulus \([E]\) might differ. Figure 12B plots the duration needed until the core oscillator’s output \([Z]\) adapted to the period of \([E]\) (time to entrainment). We define that entrainment is reached if the period length of \([Z]\) converges to those of \([E]\) up to 1'. Entrainment which demands shortening of \(Z\)’s period is more efficient than enhancing. This is plausible in the context of how PLL-based control operates: In case of a phase shift between \([E]\) and \([Z]\), the resulting error signal \([F]\) leads to a temporary acceleration of \(Z\) degradation, which in turn increases the oscillation frequency causing a shortened period. In contrast, entrainment to slower clock signals incorporates a form of inherent graduality: First, \([Z]\) temporarily increases its frequency until it converges to \([E]\) for a moment before it drifts again what causes a new adjustment of \([Z]\). The process of successive adjustment repeats consistently in a time-consuming manner.

Finally, we are going to find out the ability of our control system model to cope with both, different frequency and different initial phase between external stimulus \([E]\) and core oscillator output \([Z]\). Figure 12C picks two constant frequencies for \([E]\), one with a period length of \(24\text{h}26'\) and a second course with a period length of exactly 24h. For both courses, the time to entrainment had been observed subject to initial phase shifts between 0° and 360°, which corresponds
to a full period of the core oscillator’s output \([Z]\). It turns out that both courses reach specific minima and maxima. Its global maximum (at approx. 245°) and its global minimum (at approx. 325°) come along with a detailed consideration of period length’s variation over time towards entrainment in Figure 12D.

5 Conclusions

Exploration of chronobiological systems emerges as a growing research field within bioinformatics. From a systems biology perspective, the question arises whether biochemical control systems for regulation of oscillatory signals and their technical counterparts can utilise similar mechanisms. If so, modelling approaches adopted from building blocks are helpful to identify general components for frequency control in circadian clock systems along with gaining comprehensive insight into mechanisms of clock maintenance, synchronisation, and entrainment to external stimuli. Embedded into the concept of P modules, we presented a pure chemical model of a circadian clock system based on a phase-locked loop. Simulation studies succeeded in reproduction of its expected dynamical behaviour and its capability of entrainment. More generally, P modules deal with chemical models of analog computer components, for instance to carry out arithmetic tasks or to act as low-pass filters. Characteristic curves and transfer functions might be an advantageous choice to reduce the number of system parameters by keeping the relevant behavioural information. Further work is dedicated to enlarge the pool of P modules, e.g. by providing frequency dividers, and it will take into account a proper transformation of knowledge in systems theory into the field of membrane computing in order to provide analytic tools towards a more comprehensive understanding of chemical modules.

Acknowledgements

We gratefully acknowledge funding from the German Federal Ministry of Education and Research (project no. 0315260A) within the Research Initiative in Systems Biology.

References

12. B.A. Hawkins, H.V. Cornell (Eds.). *Theoretical Approaches to Biological Control*. Cambridge University Press, 1999


Evolutionary Design of a Simple Membrane System

Xiaoli Huang¹, Gexiang Zhang¹, and Florentin Ipate²

¹ School of Electrical Engineering, Southwest Jiaotong University, Chengdu, China
huangxiaoli-22@163.com, zhgxdylan@126.com
² Department of Computer Science, University of Pitesti, Pitesti, Romania
florentin.ipate@ifsoft.ro

Abstract. The programmability of membrane systems is an ongoing and challenging issue. This paper focuses on the automatic design of a simple membrane system for fulfilling a specific task by using a quantum-inspired evolutionary algorithm and the P-lingua simulator. The design consists of the pre-defined membrane structure and initial objects, a common set of evolution rules, the coding technique of membrane systems, evolutionary operators and a fitness function for evaluating different membrane systems. Experiments conducted on P-lingua simulator show that the presented design approach is feasible and effective to automatically evolve a membrane system for solving some specific tasks. The results also show that a quantum-inspired evolutionary algorithm is more appropriate than a genetic algorithm, recently reported in the literature, for designing a membrane system.

Keywords: Membrane computing, Membrane system, Quantum-inspired evolutionary algorithms, Evolutionary design, P-Lingua simulator

1 Introduction

As an emergent branch of natural computing, membrane computing was introduced by Gh. Păun in [1]. The main computing ideas were inspired by biochemical processes taking place inside living cells. Advances in distributed and parallel computing have motivated an intensive research effort in membrane computing.

Since the initial definition of a membrane system (or a P system) was introduced in [1] as a computing model, many different P systems have been defined and used for theoretical research as well as a vehicle to represent various problems from different fields. In [2, 3] a variant of P systems was used to solve NP-complete problems under the name of P systems with active membranes. In [4] another variant, P systems with symport/antiport, was applied for a simulation of counter automata. Reference [5] introduced a family of tissue P systems to obtain a uniform and polynomial-time solution for the satisfiability (SAT) problem. Sorting networks, which is a well-known parallel sorting model, has been implemented based on spiking neural P systems [6]. Until now, there has been much research involving various membrane computing devices,
but the programmability of membrane systems is an ongoing and challenging issue. Moreover, little work on automatic evolution of a membrane system has been reported in the literature.

This paper proposes an approach for automatically evolving a membrane system by employing a quantum-inspired evolutionary algorithm (QIEA). By pre-defining a membrane structure, initial objects and a common set of evolution rules, we use quantum-inspired bit (Q-bit) representation to encode membrane systems, and apply a quantum-inspired gate (Q-gate) to guide a membrane system toward the required model. Also a fitness function for evaluating different membrane systems is introduced. This work is different from membrane algorithms [7, 8], which combine membrane systems with evolutionary algorithms to design effective and efficient approximate optimization algorithms. Instead of binary and numeric representation, a QIEA employs Q-bit representation and fulfills its evolutionary process in a genotypic space, i.e., Q-bit space, which effectively avoids the difficulty of designing the evolutionary operators in the phenotypic space, such as crossover and mutation operators [9]. Therefore, in this paper we consider the use of a QIEA to evolve a membrane system. The feasibility and effectiveness of our approach is verified by experiments.

The paper is structured as follows: Section 2 describes the problem. Section 3 presents the evolutionary design of membrane systems in detail. The experiments and results are given in Section 4. Finally, some conclusions are drawn in Section 5.

2 Problem Description

This section starts with a brief introduction to cell-like P systems, and then the problem of designing a membrane system is described.

2.1 Cell-like P Systems

A membrane system is a computational model inspired by the functioning and structure of living cells. A cell-like P system [10] consists of three main elements: (i) a membrane structure, which is composed of several hierarchically embedded membranes; (ii) multisets of objects, occurring inside various regions delimited by each membrane in the membrane structure; and (iii) rules assigned to specified compartments of the membrane structure, processing the objects inside and the membranes themselves. The membrane system will go from one configuration to a new one by applying the rules in a nondeterministic and maximally parallel way. At the end of the computation (when no more rules can be applied), the result is obtained in a specified compartment of the system or the environment. A basic cell-like P system can be formalized as follows [10]:

\[ \Pi = (O, \mu, \omega_1, \ldots, \omega_m, R_1, \ldots, R_m, i_0) \]

where
- $O$ is an alphabet of objects;
- $\mu$ is a membrane structure with $m$ membranes and with the regions labeled by the elements of a given set $H$, $H = \{1, 2, \ldots, m\}$.
- $\omega_i$, $1 \leq i \leq m$, are strings which represent multisets over $O$ associated with the regions $1, 2, \ldots, m$ of $\mu$;
- $R_i$, $1 \leq i \leq m$, are finite sets of rules associated with the regions $1, 2, \ldots, m$ of $\mu$;
- $i_0$ is the output membrane of $\Pi$.

2.2 Problem Statement

As briefly introduced above, P systems are a class of multi-tuple and flexible models. In this paper we only consider the basic type of P systems, the cell-like P systems. Each P system use a specific membrane structure, objects and rules depending on the problem it works on. When a P system is designed for solving a task, a researcher usually has an approximate idea at the initial stage [11]. To be more specific, in most cases the membrane structure, the multisets of objects and the alphabet of objects can be preliminarily determined, although finding specific and feasible rules is not an easy task.

In our study, given the membrane structure, the multisets of objects, the alphabet of objects and a set of (redundant) rules that can be used for solving a class of problems, the main aim is to use an evolutionary algorithm to find a proper set of rules, by appropriately evolving a membrane system for a specific task. The problem can be summarized as follows:

An initial configuration including a membrane structure $\mu$ and its set of labels $H = \{1, 2, \ldots, m\}$, initial multisets $W = \{\omega_1, \omega_2, \ldots, \omega_m\}$ and an alphabet of objects $O$ is given. We consider a family of P systems $\Pi = \{\Pi_i\}_{i \in I}$, where $\Pi_i = (O, H, \mu, W, R_i)$ is a P system represented as a 5-tuples and $I$ is a set of natural numbers, in which $O$ is the alphabet of objects; $H = \{1, 2, \ldots, m\}$ is the set of labels; $\mu$ is the membrane structure; $W = \{\omega_1, \omega_2, \ldots, \omega_m\}$ is the initial multisets and $R_i = \{r_1, r_2, \ldots, r_l\}_{l \in I}$ is the set of rules. These P systems in the family are regarded to have the same initial configuration, including the same $O, H, \mu$ and $W$. The sole difference between these P systems is their set of rules $R_i$. In order to obtain a formal definition of the family of P systems $\Pi$, we will define a common set of rules $\mathcal{R}$, from which different rules are taken and combined for different P systems. More specifically, let the finite common set of rules be $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}_{k \in I}$; the set of rules $R_i \subseteq \mathcal{R}$ is a subset of $\mathcal{R}$. Then we can employ a QIEA to design a P system. In this algorithm, an initial population of P systems is generated by combining different rules which are taken from the predefined common set of rules $\mathcal{R}$, and then such a population will evolve according to the mechanism of QIEAs.

This work considers the following two types of rules in $\mathcal{R}$ [11]:

**Evolution rules:** $[d \rightarrow s]_m$. In the membrane with label $m$ one copy of object $d$ is involved in the rule and evolves to the multiset $s$.

**Dissolution rules:** $[d]_m \rightarrow s$. A membrane labeled with $m$ is dissolved in reaction with one copy of object $d$, meanwhile the object $d$ transforms
into object $s$ and all the objects and rules in the membrane $m$ go to the surrounding region.

Additionally, the evolution of all the P systems represented by individuals of a QIEA is considered to be deterministic, which ensures that there is only one rule triggered by the same object for each membrane in each P system.

### 3 Evolutionary Design

In what follows, we present our approach that uses a QIEA to design a P system, which performs a pre-assigned task, from an initial population of P systems. The evolution of quantum-inspired bit (Q-bit) individuals will only correspond to the changes within the set of rules $R_i$ of each P system. In the process of evolution, a proper fitness function needs to be defined to guide Q-bit individuals toward to the required P system. The pseudocode algorithm of QIEA is shown in Fig.1 [12].

<table>
<thead>
<tr>
<th>Algorithm for QIEA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Begin</strong></td>
</tr>
<tr>
<td>$t \leftarrow 1$</td>
</tr>
<tr>
<td>1) Generate an initial population $Q(t)$ represented by Q-bit individuals;</td>
</tr>
<tr>
<td><strong>While</strong> (not termination-condition) <strong>do</strong></td>
</tr>
<tr>
<td>2) Make a binary population $X(t)$ by observing the states of $Q(t)$;</td>
</tr>
<tr>
<td>3) Evaluate all the individuals in $X(t)$;</td>
</tr>
<tr>
<td>4) Update $Q(t)$ using Q-gates and produce offspring;</td>
</tr>
<tr>
<td>$t \leftarrow t + 1$</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

**Fig. 1. The pseudocode algorithm of QIEA**

In the pseudocode algorithm of QIEA, $t$ represents the evolutionary generation. Based on the pseudocode algorithm of QIEA, the evolutionary design of a P system can be described as follows:

1) **Population initialization.** In order to generate the initial population, $n$ different random subsets of a common set of rules $\mathcal{R} = \{r'_1, r'_2, \ldots, r'_k\}_{k \in I}$ are considered, where $n$ is the number of individuals in the population and $k$ is the number of rules in the $\mathcal{R}$. The population at generation $t$ is defined as $Q(t) =$
\( \{q^t_1, q^t_2, \ldots, q^t_n\} \), where \( q^t_j \) \((j = 1, 2, \ldots, n)\) is a Q-bit individual defined as (1):

\[
q^t_j = \begin{bmatrix}
\alpha^t_{j1} & \alpha^t_{j2} & \cdots & \alpha^t_{jk} \\
\beta^t_{j1} & \beta^t_{j2} & \cdots & \beta^t_{jk}
\end{bmatrix}
\]  

A P system mentioned in Sect. 2, which is encoded by an individual, can be represented by a string of \( k \) Q-bits as (1), where each Q-bit corresponds to a rule in the common rule set \( \mathcal{R} \); the observed value of each Q-bit, 1 or 0, indicates that the corresponding rule will be selected or not for the associated P system.

2) Observation. The step of "observing the states of \( Q(t) \)" is used to translate the Q-bit individuals into binary solutions. By observing the population \( Q(t) \), QIEA can get a binary population \( X(t) = \{X^t_1, X^t_2, \ldots, X^t_n\} \). The binary solution \( X^t_j \) \((j = 1, \ldots, n)\) is a binary string of length \( k \), which is formed from 0 or 1 for each bit by using the probability. This process of probabilistic observation is shown in Fig.2, in which \( x \) is the observed value of the Q-bit shown as (2).

\[
\begin{bmatrix}
\alpha \\
\beta
\end{bmatrix}
\]

(2)

The observation process

<table>
<thead>
<tr>
<th>The observation process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>If ( \text{random} [0, 1] &lt; [\alpha]^T )</td>
</tr>
<tr>
<td>Then ( x \leftarrow 0 )</td>
</tr>
<tr>
<td>Else ( x \leftarrow 1 )</td>
</tr>
<tr>
<td>End</td>
</tr>
</tbody>
</table>

Fig. 2. Observation process in the QIEA [12]

3) Evaluation. Following the Q-bit representation of P systems, we define a fitness function to evaluate each possible P system (solution or individual). The fitness value of an individual depends on the absolute value of the difference between the corresponding simulation result obtained from the P-Lingua simulator and the expected result for the pre-assigned computational task. In this work the result is defined as a number of the specified objects present in a certain membrane in the halting configuration of a P system. Thus the fitness function is:

\[
\text{fitness} = |\text{the simulation result} - \text{the expected computing result}|
\]
From the evaluation’s point of view, the most important difficulty for evaluating an individual is the simulation of a P system based on the P-Lingua simulator [13]. P-Lingua is a programming language for membrane computing. A Java library called pLinguaCore has been developed as a software framework for cell-like P systems. In this paper we use pLinguaCore to perform all the simulations involved in the evolutionary design of P systems. The implementation procedure of a P system on the P-Lingua simulator is shown in Fig.3. More details about P-Lingua are available on the P-Lingua web page [14].

![Fig. 3. The implementation procedure of a P system on P-Lingua simulator](image)

As a practical application of the P-Lingua simulator in our experiments, we create a corresponding real-time P-Lingua file which defines each P system in the P-Lingua language; then we compile the P-Lingua file by using the P-Lingua compiler and implement the simulation of the P system. The simulator will perform the computation that the P system may follow and generate a text file with a detailed step-by-step report of the computation [15]. Using these, the evaluation for each P system can be fulfilled.

The fitness value of each individual will be computed in a real-time way. The fitness 0 indicates that we have found a P system which is capable of performing the pre-assigned task and the evolutionary design has been successful.

4) **Update.** At the "update $Q(t)$" step, all the individuals in $Q(t)$ are updated by Q-gates. In this paper, we use the quantum rotation gate in [12] to perform this step, the rotation gate is operated on each Q-bit. To be specific, the $i$th Q-bit in the $j$th Q-bit individual $q^t_{ji}$, $i = 1, 2, \ldots, k$; $j = 1, 2, \ldots, n$, is updated by applying the current Q-gate $G^t_{ji} (\theta)$

$$G^t_{ji} (\theta) = \begin{bmatrix} \cos \theta^t_{ji} & -\sin \theta^t_{ji} \\ \sin \theta^t_{ji} & \cos \theta^t_{ji} \end{bmatrix}$$

(3)

where $\theta^t_{ji}$ is an adjustable Q-gate rotation angle, the selective strategy of $\theta^t_{ji}$ is shown as the Table I in [12]. Thus, the update procedure for the Q-bit $[\alpha^t_{ji} \beta^t_{ji}]^T$
can be described as
\[
\begin{bmatrix}
\alpha_{ji}^{t+1} \\
\beta_{ji}^{t+1}
\end{bmatrix} = G_{ji}^t(\theta)
\begin{bmatrix}
\alpha_{ji}^t \\
\beta_{ji}^t
\end{bmatrix}
\] (4)

4 Experiments and Results

In this section, experiments are performed to design two simple P systems by using the evolutionary design approach presented in Sect. 3. All the experiments are implemented on the platform Eclipse 3.5.1 and JDK 1.6.0 by using a laptop with 1.8GHz, 768MB RAM and Windows XP OS.

4.1 An example for computing $n^2$

The first experiment focuses on designing a specific P system which computes the square of any natural number $n \geq 2$. The initial configuration and the common set of rules $\mathcal{R}$ are designed as follows:

(a) The initial configuration is $\Pi = (O,H,\mu,w_1,w_2,R)$, where $O = \{a,b,c,d,f\}$ is the alphabet, the set of labels $H = \{1,2\}$, the membrane structure $\mu = [\square_2]_1$ and the initial multisets $w_1 = \emptyset$ and $w_2 = a^2bd$.

(b) The set of rules $\mathcal{R}$ is defined as follows ($n \geq 2$ is a natural number, $\lambda$ is the empty multiset):

\[
\mathcal{R} = \begin{cases}
    r'_1 \equiv [a \rightarrow ab]_2 & r'_7 \equiv [c]_2 \rightarrow a & r'_{13} \equiv [b \rightarrow \lambda]_1 \\
    r'_2 \equiv [b \rightarrow bc]_2 & r'_8 \equiv [b \rightarrow c]_2 & r'_{14} \equiv [d \rightarrow \lambda]_1 \\
    r'_4 \equiv [c]_2 \rightarrow b & r'_{10} \equiv [a^2 \rightarrow c]_2 & r'_{15} \equiv [f \rightarrow \lambda]_1 \\
    r'_5 \equiv [a \rightarrow bc]_2 & r'_{11} \equiv [d \rightarrow df]_2 & r'_{16} \equiv [f^{n}]_2 \rightarrow \lambda \\
    r'_6 \equiv [b^2 \rightarrow c]_2 & r'_{12} \equiv [a \rightarrow \lambda]_1 & r'_{17} \equiv [f^{(n+1)}]_2 \rightarrow \lambda \\
    r'_9 \equiv [a \rightarrow \lambda]_1 & r'_{13} \equiv [b \rightarrow \lambda]_1 & r'_{18} \equiv [f^{(n-1)}]_2 \rightarrow \lambda
\end{cases}
\]

In this experiment, the parameter setting for QIEA is as follows: the population size is 30 and the maximal number of independent runs is 30. In order to generate the initial population, we consider 30 different random subsets of $\mathcal{R}$, and the length of each individual is 18, which is same as the number of rules in the $\mathcal{R}$.

In our experiment the computing result of each P system is the number of $c$ objects in the skin membrane in the halting configuration. As $n$ is a variable, the fitness function is dynamic and represented as $|\text{the simulation result} - n^2|$. When we compute the squares of different numbers $n$, the fitness function will vary with the value of $n$. To avoid a non-ending computation, the maximal number of steps allowed for each computational simulation of P systems is set to ($n + 5$), which also changes with the value of $n$. Specifically, we generate a new random number $n$ before the evaluation and simulation of each P system, and then the parameter value of $n$ will be passed on to the corresponding fitness
function, the maximal number of steps allowed and the rules in the P system. In this experiment the variable \( n \) can vary from 2 to 100.

After performing 30 independent runs, the number of successful runs, in which we evolve a correct P system to solve the square problem, is obtained. The number of successful runs, the average number of evolutionary generations required and the computing time per run are shown in Table 1. The experimental results show that 21 out of the 30 runs are successful, the remaining 9 runs fail to obtain any satisfactory P systems for fulfilling the computing task. Out of 21 successful runs, 12 satisfactory P systems \( (P_{\text{sat}}) \) are distinct; their sets of rules \( R_{\text{sat}} \) are described as follows:

\[
R_{\text{sat}1} = \begin{cases} 
[a \to ab]_2 & [d \to \lambda]_1 \\
[b \to bc]_2 & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda \\
b \to \lambda & 1 
\end{cases} \quad R_{\text{sat}2} = \begin{cases} 
[a \to ab]_2 & [b \to \lambda]_1 \\
b \to bc & [d \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda \\
a \to \lambda & 1 
\end{cases}
\]

\[
R_{\text{sat}3} = \begin{cases} 
[a \to ab]_2 & [d \to \lambda]_1 \\
b \to bc & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda \\
a \to \lambda & 1 
\end{cases} \quad R_{\text{sat}4} = \begin{cases} 
[a \to ab]_2 & [b \to \lambda]_1 \\
b \to bc & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda \\
a \to \lambda & 1 
\end{cases}
\]

\[
R_{\text{sat}5} = \begin{cases} 
[a \to ab]_2 & [a \to \lambda]_1 \\
b \to bc & [d \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda 
\end{cases} \quad R_{\text{sat}6} = \begin{cases} 
[a \to ab]_2 & [b \to \lambda]_1 \\
b \to bc & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda 
\end{cases}
\]

\[
R_{\text{sat}7} = \begin{cases} 
[a \to ab]_2 & [d \to \lambda]_1 \\
b \to bc & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda 
\end{cases} \quad R_{\text{sat}8} = \begin{cases} 
[a \to ab]_2 & [a \to \lambda]_1 \\
b \to bc & [f \to \lambda]_1 \\
d \to df & [f^{(n-1)}]_2 \to \lambda 
\end{cases}
\]

\[
R_{\text{sat}9} = \begin{cases} 
[a \to ab]_2 & [d \to \lambda]_1 \\
b \to bc & [f^{(n-1)}]_2 \to \lambda \\
d \to df & 2 
\end{cases} \quad R_{\text{sat}10} = \begin{cases} 
[a \to ab]_2 & [a \to \lambda]_1 \\
b \to bc & [f^{(n-1)}]_2 \to \lambda \\
d \to df & 2 
\end{cases}
\]

\[
R_{\text{sat}11} = \begin{cases} 
[a \to ab]_2 & [f \to \lambda]_1 \\
b \to bc & [f^{(n-1)}]_2 \to \lambda \\
d \to df & 2 
\end{cases} \quad R_{\text{sat}12} = \begin{cases} 
[a \to ab]_2 & [d \to df]_2 \\
b \to bc & [f^{(n-1)}]_2 \to \lambda \\
d \to df & 2 
\end{cases}
\]

### 4.2 Comparison with genetic algorithms

The second experiment aims to design a P system for computing the square of number 4; this problem has been tackled by using a genetic algorithm (GA), as
In order to compare our approach with GAs, this experiment uses the same initial configuration and set of rules $\mathcal{R}$:

(a) The initial configuration is $H = (O,H,\mu, w_1, w_2, R)$, where $O = \{a, b, c, z_1, z_2, z_3, z_4\}$ is the alphabet, the set of labels $H = \{1, 2\}$, the membrane structure $\mu = [\{\}]_2$, and the initial multisets $w_1 = \emptyset$ and $w_2 = a^2b^2z_1$.

(b) The set of rules $\mathcal{R}$ is:

$$
\mathcal{R} = \left\{
\begin{array}{c}
r'_1 \equiv [a \rightarrow ab]_2 \\
r'_2 \equiv [b \rightarrow bc]_2 \\
r'_3 \equiv [c \rightarrow b^2]_2 \\
r'_4 \equiv [a \rightarrow bc]_2 \\
r'_5 \equiv [z_1 \rightarrow z_2]_2 \\
r'_6 \equiv [z_2 \rightarrow z_3]_2 \\
r'_7 \equiv [z_2 \rightarrow z_1]_2 \\
r'_8 \equiv [z_3 \rightarrow z_4]_2 \\
r'_9 \equiv [b \rightarrow \lambda]_1 \\
r'_{10} \equiv [z_2 \rightarrow a]_2 \\
r'_{11} \equiv [z_3 \rightarrow c]_2 \\
r'_{12} \equiv [z_4 \rightarrow a]_2 \\
r'_{13} \equiv [a \rightarrow \lambda]_1 \\
r'_{14} \equiv [b \rightarrow \lambda]_1 \\
r'_{15} \equiv [b \rightarrow c]_2 \\
r'_{16} \equiv [c \rightarrow \lambda]_2 \\
r'_{17} \equiv [z_4 \rightarrow z_1]_2 \\
r'_{18} \equiv [z_4 \rightarrow b]_2
\end{array}\right\}
$$

A QIEA is used to evolve a P system computing the square of number 4, from an initial population of P systems. The computing result of each P system is given in the same form as in the previous experiment i.e., the number of $c$ objects in the skin membrane in the halting configuration; the expected result is obviously 16 copies of $c$ objects; the fitness function is $|\text{the simulation result} - 16|$. We use 15 as the maximal number of steps allowed for each computational simulation of P systems.

The parameter setting for QIEA is the same as in the first example. The number of successful runs out of 30 independent runs, the average number of evolutionary generations required and the computing time per run are given in Table 1. The best experimental result obtained by GAs in [11] is also presented in Table 1. Out of the 12 successful runs by QIEA, 7 satisfactory P systems ($P_{sat}$) are distinct and their sets of rules $R_{sat}$ are described as follows:

$$
P_{sat1} = \left\{ [a \rightarrow ab]_2, [z_2 \rightarrow z_3]_2, [z_1 \rightarrow z_2]_2 \right\} \quad P_{sat2} = \left\{ [a \rightarrow ab]_2, [z_2 \rightarrow z_3]_2, [z_1 \rightarrow z_2]_2, [z_4]_2 \rightarrow a \right\}
$$

$$
P_{sat3} = \left\{ [a \rightarrow ab]_2, [z_3 \rightarrow z_4]_2, [z_1 \rightarrow z_2]_2, [z_2 \rightarrow z_3]_2 \right\} \quad P_{sat4} = \left\{ [a \rightarrow ab]_2, [z_3 \rightarrow z_4]_2, [z_1 \rightarrow z_2]_2, [z_4]_2 \rightarrow a, [b \rightarrow bc]_2, [a]_1 \rightarrow [b]_1 \right\}
$$

$$
P_{sat5} = \left\{ [a \rightarrow ab]_2, [z_3 \rightarrow z_4]_2, [z_1 \rightarrow z_2]_2, [z_2 \rightarrow z_3]_2 \right\} \quad P_{sat6} = \left\{ [a \rightarrow ab]_2, [z_3 \rightarrow z_4]_2, [z_1 \rightarrow z_2]_2, [z_4]_2 \rightarrow b, [b \rightarrow bc]_2, [a]_1 \rightarrow [b]_1 \right\}
$$
\[ R_{sat7} = \begin{cases} 
[a \rightarrow ab]_2 & [z_3 \rightarrow z_4]_2 \\
[b \rightarrow bc]_2 & [b \rightarrow \lambda]_1 \\
[z_1 \rightarrow z_2]_2 & [z_4]_2 \rightarrow a \\
[z_2 \rightarrow z_3]_2 
\end{cases} \]

**Table 1.** Experimental results (The symbol ‘–’ means that the information was not provided in [11])

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Successful runs</th>
<th>Average generations</th>
<th>Time per run(seconds)</th>
</tr>
</thead>
</table>
| QIEA &isub;2 & 21/30 & 18.47 & 2.97 &
| QIEA &isub;4 & 12/30 & 20.97 & 3.13 &
| GA &isub;4 & 1/30 & – & – &

The experimental results obtained from the two examples show that the introduced approach is feasible and effective to evolve a P system for performing a simple mathematical operation. Compared with the results in [11], we obtain much more successful runs; this improved performance is a consequence of the advantages of QIEAs outlined above. It is worth noting that since QIEAs do not use mutation operators, it seems that the resulting search space may be considerably smaller. The mutation operation in [11] might expand the search space, but its role is not clear, due to the loss in performance.

## 5 Conclusions

This paper presented a clear possibility on how to use QIEAs to design a desired P system, including the coding, evolutionary process and evaluation function. QIEAs use the Q-bit representation to encode membrane systems and apply the Q-gate as an evolutionary operator to guide a membrane system toward the required model, instead of utilizing a crossover or a mutation operator as those in GAs. As compared with GAs, QIEAs are simpler and therefore more appropriate for designing a P system. The experimental results from two examples show that the approach is very promising to automatically evolve a membrane system by employing a computer program. In further works, we will generalize this approach to a more general or complex class of membrane systems. In the presented method, the design of the set of rules and the fitness function are very important. Therefore, much attention will be devoted to them in future works.

**Acknowledgment**

This work is supported by the Scientific and Technological Funds for Young Scientists of Sichuan (09ZQ026-040), the Fund of Engineering Research Centre
for Transportation Safety, Ministry of Education (WHUTERCTS2010A01), the
Fundamental Research Funds for the Central Universities(SWJTU09ZT10) and
the Scientific Research Foundation for the Returned Overseas Chinese Scholars,
State Education Ministry.

References

1. Păun, Gh.: Computing with membranes. Journal of Computer and System Sci-
ces. 61(1), 108–143 (2000)
2. Krishna, S.N., Rama, R.: A variant of P systems with active membranes: Solving
NP-complete problems. Romanian J. of Information Science and Technology. 2(4),
357–367 (1999)
J. Automat. Lang. Comb. 6(1), 75–90 (2001)
based on P systems for a class of combinatorial optimization. Fundamenta Informaticae.
87(1), 93–116 (2008)
brane algorithms. Mathematical and Computer Modelling. 52(11–12), 1997–2010
(2010)
Science. 287(1), 73–100 (2002)
11. Escuela, G., Gutiérrez-Naranjo, M.A.: An application of genetic algorithms to
12. Han, K.H., Kim, J.H.: Quantum-inspired evolutionary algorithm for a class of
overview of P-Lingua 2.0. Lecture Notes in Computer Science. vol. 5957, 264–288
(2010)
15. Díaz-Pernil, D., Pérez-Hurtado, I., Pérez-Jiménez, M.J., et al.: A P-Lingua Pro-
gramming Environment for Membrane Computing. Lecture Notes in Computer
Periodicity as a Dynamical Aspect of Generative Spiking Neural P Systems

Gyenn Neil Ibo, Henry Adorna

Algorithms & Complexity Lab
Department of Computer Science
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines
E-mail: gvibo@up.edu.ph, hnadorna@dcs.upd.edu.ph

Abstract. In this paper, we explored periodicity as a particular dynamical aspect in the study of generative Spiking Neural P Systems. Taking a cue from the role of periodicity in nature, we highlighted this aspect in SN P systems by providing a simple formal definition and a necessary condition via the matrix representation formulated in a previous paper, together with some other concepts from linear algebra. We then accordingly examined particular examples of generative systems from the seminal paper that introduced SN P systems, took note of their periodic behavior, and in the process empirically verified the concepts and theorems we presented. We also noted that these algebraic illustrations, aside from being an elegant representation from a theoretical point of view, will also be instrumental in providing some sort of a design framework for SN P systems.

Key words: Membrane Computing, Spiking Neural P Systems, Matrix Representation, Periodicity, Dynamical Aspects of P Systems

1 Introduction

Spiking Neural P Systems were introduced in [4] as a particular type of P System that abstracts and applies ideas from neurobiology. This system consists of mono-membranar cells, with the synaptic connections in between them serving as media for transporting an object called a spike. Despite several restrictive characteristics of these systems, they have been found to be Turing complete [4].

Among the many peculiarities of SN P systems, the idea of encoding information as time duration is the most distinctive. These systems are designed such that their computations rely heavily on the amount of time elapsed between particular events, and their output is likewise represented as such. This idea is derived from the fact that most of the neural impulses are almost identical, electrical signals of a given voltage [4]. Information, thus, is not encoded in the strength of these signals, but in the frequency and the time of their occurrence.

In recent years, quite a number of topics have been raised [3] and studied in this relatively young field. In [5], for instance, the already formal SN P systems
are further formalized and abstracted. The result of this particular study is an algebraic representation of the concepts in SN P systems: the configuration of a system, the set of rules that can be applied, and the amount of spikes gained and lost by neurons are represented as vectors, the structure and relation of the rules to the neurons is represented as a matrix, and the event of a computation, or the application of the rules, is represented by multiplying a vector with a matrix. Aside from the fact that it simplifies the programmers job of coding a program simulating SN P systems, what is more admirable from a theoretical viewpoint is that it presents an elegant representation of a rather complex system.

In this paper, we make use of the foundations laid in [5] to explore a particular dynamical aspect of SN P systems periodicity. Dynamical studies and investigations on periodicity have already been conducted in other types of P Systems [1]. We aim to do the same in this paper, albeit quite differently in the sense that we explore it through the context of the algebraic representations introduced in [5].

The paper is organized as follows. In the next section, we present the formal definition of SN P systems. In Section 3, we recall the concepts and definitions in [5]. Sections 4 and 5 contain the main results of this paper. In the former, we presented some additional definitions and lemmas, and provided a necessary condition for SN P systems to be periodic. Here we considered the SN P systems in generative mode. In the latter, we adapted some examples of generative SN P systems presented in [4], and carefully inspected them so as to empirically verify our claims in Section 4. We end with the final remarks in Section 6.

2 Spiking Neural P Systems

We present here the formal definition of SN P systems without delay, which is a restricted variant, as adapted from [5].

Definition 1 (SN P systems without delay). An SNP system without delay, of degree \( m \geq 1 \), is a construct of the form

\[
\Pi = (O, \sigma_1, \ldots, \sigma_m, \text{syn, in, out}),
\]

where:

1. \( O = \{a\} \) is the alphabet made up of only one object \( a \), called spike;
2. \( \sigma_1, \ldots, \sigma_m \) are \( m \) neurons of the form

\[
\sigma_i = (n_i, R_i), 1 \leq i \leq m,
\]

where:

(a) \( n_i \geq 0 \) gives the initial number of spikes \( (a) \) contained in neuron \( \sigma_i \);
(b) \( R_i \) is a finite set of rules of the following forms:

(b-1) \( E/a^c \rightarrow a^p \), are known as Spiking rules, where \( E \) is a regular expression over \( a \), and \( c \geq 1 \), for \( p \geq 1 \) number of spikes are produced (with the restriction \( c \geq p \)), transmitted to each adjacent neuron with \( \sigma_i \) as the originating neuron, and \( a^c \in L(E) \);
(b-2) \( a^s \rightarrow \lambda \), are known as Forgetting rules, for \( s \geq 1 \), such that for each rule \( E/a^c \rightarrow a^p \) of type (b-1) from \( R_i \), \( a^s \notin L(E) \);

3. \( \text{syn} = \{(i, j) \mid 1 \leq i, j \leq m, i \neq j\} \) are the synapses i.e. connections between neurons;

4. \( \text{in}, \text{out} \in \{1, 2, \ldots, m\} \) are the input and output neurons, respectively.

The rules of type (b-1) can be applied if the neuron \( \sigma_i \) contains \( k \) spikes, and \( a^k \in L(E) \), \( k \geq c \). Applying this type of rule consumes \( c \) spikes from neuron \( \sigma_i \), and sends \( p \) spikes to all neurons to which it has an outgoing synapse.

The type (b-2) rules (also known as forgetting rules) can be applied if neuron \( i \) contains exactly \( s \) spikes; this removes all the spikes in the neuron.

During each time unit, if a neuron \( \sigma_i \) can apply one of its rules, then a rule from \( R_i \) has to be applied. Note that it is possible that two or more rules in \( R_i \) can be validly applied at a particular time unit, in which case only one of them is non-deterministically chosen and applied.

This means that the rules are applied in a sequential manner in each neuron, at most one at a time, whereas several neurons can fire simultaneously, functioning in parallel.

The number of spikes in each neuron represents the configuration of the system during that time-step. For each configuration, a particular set of rules, according to the criteria described above, can be applied. This produces a sequence of configurations, which is called a computation of the system. This computation halts if it reaches a configuration where no rule can be applied. The number of steps elapsed between the first two spikes of the designated output neuron is considered as the output of the system.

3 Matrix Representation of SN P Systems

We recall here the matrix representation of SN P systems introduced in [5]. We refer the reader to the said paper for further details.

**Definition 2 (Configuration Vector).** Let \( \Pi \) be an SN P system with \( m \) neurons, the vector \( C_0 = (n_1, n_2, \ldots, n_m) \) is called the initial configuration vector of \( \Pi \), where \( n_i \) is the amount of initial spikes present in neuron \( \sigma_i \), \( i = 1, 2, \ldots, m \) before the computation starts.

For any \( k \in \mathbb{N} \), the vector \( C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)}) \) is called the \( k \)th configuration vector of the system, where \( n_i^{(k)} \) is the amount of spikes in neuron \( \sigma_i \), \( i = 1, 2, \ldots, m \) after the \( k \)th step in the computation.

**Definition 3 (Spiking Vector).** Let \( \Pi \) be an SN P system with \( m \) neurons and \( n \) rules. Assume a total order \( d : 1, n \) is given for all the \( n \) rules, so the rules can be referred as \( r_1, \ldots, r_n \). Moreover, let \( C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)}) \) be the configuration at time-step \( k \), and \( E_1, \ldots, E_n \) be the corresponding regular expressions defined in Definition 1. A spiking vector \( s^{(k)} \) is defined as \( s^{(k)} = (r_1^{(k)}, r_2^{(k)}, \ldots, r_n^{(k)}) \).
Note: The spiking vector $s^{(k)}$ is computed as follows: $r_i^{(k)}$ is assigned as 1 if the amount of spikes $n_i^{(k)}$ in neuron $i$ at time-step $k$ satisfies the regular expression $E_i$, 0 otherwise.

The spiking transition matrix in the following definition will represent the amount of spikes consumed (or received) by each neuron in every application of each rule.

**Definition 4 (Spiking Transition Matrix).** Let $\Pi$ be an SN P system with $m$ neurons and $n$ rules, and $d:1\ldots n$ be a total order for all the $n$ rules. The spiking transition matrix $M_\Pi$ of the system $\Pi$ is defined as follows:

$$M_\Pi = [a_{ij}]_{nxm},$$

where $a_{ij}$ is equal to the following:

- $-c$, if rule $r_i$ is in neuron $\sigma_j$ and it is applied consuming $c$ spikes;
- $p$, if rule $r_i$ is in neuron $\sigma_s$ ($s \neq j$ and $(s, j) \in \text{syn}$) and it is applied producing $p$ spikes;
- $0$, if rule $r_i$ is in neuron $\sigma_s$ ($s \neq j$ and $(s, j) \notin \text{syn}$).

**Definition 5 (Transition Net Gain Vector).** Let $\Pi$ be an SN P system with $m$ neurons and $n$ rules, and $C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)})$ be the $k$th configuration vector of $\Pi$. The transition net gain vector at step $k$ is defined as:

$$NG^{(k)} = C_{k+1} - C_k$$

Note that the transition net gain vector $NG^{(k)}$ can also be derived as the product between the spiking vector $s^{(k)}$ and the spiking transition matrix $M_\Pi$, as stated in Lemma 3.1 in [5]. That is,

$$NG^{(k)} = s_k \cdot M_\Pi$$

### 4 Periodic SN P Systems

Here we examine periodicity as a crucial aspect of generative SN P systems. In subsection 4.1, we add our own concepts and definitions to the ones presented in Section 3. These concepts enable us to provide a simple but formal definition of periodicity. In subsection 4.2, we used these concepts and definitions to formulate a necessary condition for SN P systems to be periodic.

Note that in order to lay the foundation for defining a periodic SN P System, here we initially consider systems with only one rule per neuron. This will give us a spiking transition matrix $M_\Pi$ that has as many rows as there are columns; thus, the matrix is square. Furthermore, we will relate this with the general generative SN P systems we encounter in Section 5.
4.1 Definitions

Definition 6 (Computation Sequence). Let $\Pi$ be an SN P system with $m$ neurons and $m$ rules (one rule for each neuron), $C_0 = (n_1^{(0)}, n_2^{(0)}, \ldots, n_m^{(0)})$ be the initial configuration vector, $s^{(0)} = (r_1^{(0)}, r_2^{(0)}, \ldots, r_m^{(0)})$ be the initial spiking vector, and $M_{\Pi}$ be the spiking transition matrix of $\Pi$. The computation sequence of $\Pi$ is a sequence $COMP_{\Pi}seq$ consists of configuration vectors (as defined in Definition 2) starting with $C_0$, with the proceeding elements obtained recursively by the formula:

$$C_k = C_{k-1} + s^{(k-1)} \cdot M_{\Pi}$$

Equation 3 is adapted from Theorem 3.1 in [5]. The spiking vectors $s^k$ can be determined from the configurations $C_k$, as described in the note right after Definition 3.

Definition 7 (Periodic SN P system). Let $\Pi$ be an SN P system with $m$ neurons and $m$ rules (one rule for each neuron), and $COMP_{\Pi}seq$ be the computation sequence of $\Pi$. The system $\Pi$ is periodic if and only if there exist configurations $C_k$ and $C_p$ in $COMP_{\Pi}seq$, $k \neq p$, and $C_k = C_p$. That is, an SN P System is periodic (or ultimately periodic) if and only if a particular configuration is repeated in the sequence $COMP_{\Pi}seq$.

Note: If the initial configuration $C_0$ is included in the repeated configurations, then the system is periodic; otherwise, it is ultimately periodic. But in our definition of periodic SN P systems here, we do not distinguish between the two cases; we consider both to be periodic.

Take note that $\Pi$ has exactly one rule per neuron (m rules and m neurons). This makes it a deterministic system, since in every configuration there is no ambiguity of which rule to be chosen to apply in any neuron. In any deterministic system, from any given configuration $C_k$, there is only possible configuration $C_{k+1}$ to which the system transitions. Thus, if the $k$th element $C_k$ in the computation sequence $COMP_{\Pi}seq$ is later repeated in the $p$th element $C_p$, then the subsequence $(C_{k+1}, C_{k+2}, C_{p-1})$ that immediately follows $C_k$ and immediately precedes $C_p$ will also be the same subsequence $(C_{p+1}, C_{p+2}, C_{s-1})$, that immediately follows $C_p$ and immediately precedes $C_s$, where $s = p + (p - k)$, and $C_s = C_p = C_k$. Likewise, the same subsequence will also immediately follow $C_s$, and so on. This forms a periodic sequence, where the period is $(p - k)$.

Moreover, given that a system $\Pi$ is periodic, then it implies that at least one configuration $C_k$ during a particular time step $k$ must be repeated at some other time-step $p$.

Definition 8 (Spiking Vector Sequence). Let $\Pi$ be an SN P system with $m$ neurons and $m$ rules (one rule for each neuron), and $COMP_{\Pi}seq = (C_0, C_1, C_2, \ldots)$ be the computation sequence (where each $C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)})$) of $\Pi$. Assume a total order $d : 1, \ldots, n$ is given for all the $m$ rules, so the rules can be referred as $r_1, \ldots, r_m$, and $E_1, \ldots, E_m$ the corresponding regular expressions defined in
Definition 1. The spiking vector sequence \( SV_{seq} \) is a sequence that consists of spiking vectors (as defined in Definition 3) \( s^{(k)} = (r_1^{(k)}, r_2^{(k)}, \ldots, r_m^{(k)}) \), where \( s^{(k)} \) is the spiking vector corresponding to every configuration \( C_k \) in the COMP\( \text{seq} \). This is the sequence of all the spikings during the lifespan of the system \( \Pi \).

Note: The spiking vector \( s^{(k)} \) is computed as follows: \( r_i^{(k)} \) is assigned as 1 if the amount of spikes \( n_i^{(k)} \) in neuron \( i \) at time-step \( k \) satisfies the regular expression \( E_i \), 0 otherwise.

Definition 9 (Aggregate Spiking Vector). Let \( \Pi \) be an SN P system with \( m \) neurons and \( m \) rules (one rule for each neuron), and \( SV_{\text{seq}} = (s^{(0)}, s^{(1)}, s^{(2)}, \ldots) \) be the spiking vector sequence of \( \Pi \). The aggregate spiking vector \( (SV)^{k+v}_k \) is a summation of any finite consecutive subsequence of \( SV_{\text{seq}} \), from the \( k \)th to the \( (k+v) \)th element.

Given \( (s^{(k)}, s^{(k+1)}, \ldots, s^{(k+v)}) \) as a finite consecutive subsequence of \( SV_{\text{seq}} \), the aggregate spiking vector \( (SV)^{k+v}_k \) is given by the summation \( \sum_{i=k}^{k+v} s^{(i)} \).

Definition 10 (Transition Net Gain Vector Sequence). Let \( \Pi \) be an SN P system with \( m \) neurons and \( m \) rules (one rule for each neuron), and \( SV_{\text{seq}} = (s^{(0)}, s^{(1)}, s^{(2)}, \ldots) \) be the spiking vector sequence of \( \Pi \). The transition net gain vector sequence \( NG_{\text{seq}} \) is a sequence that consists of transition net gain vectors \( NG^{(k)} \), obtained by the formula \( NG^{(k)} = s^{(k)} \cdot M_{\Pi} \) (from Equation 2) for every \( s^{(k)} \) in the spiking vector sequence \( SV_{\text{seq}} \). This is the sequence of all the transition net gain vectors during the lifespan of the system \( \Pi \).

Definition 11 (Aggregate Transition Net Gain Vector). Let \( \Pi \) be an SN P system with \( m \) neurons and \( m \) rules (one rule for each neuron), and \( NG_{\text{seq}} = (NG^{(0)}, NG^{(1)}, NG^{(2)}, \ldots) \) be the transition net gain vector sequence of \( \Pi \). The aggregate transition net gain vector \( (NG)^{k+v}_k \) is a summation of any finite consecutive subsequence of \( NG_{\text{seq}} \), from the \( k \)th to the \( (k+v) \)th element.

Given \( (NG^{(k)}, NG^{(k+1)}, \ldots, NG^{(k+v)}) \) as a finite consecutive subsequence of \( NG_{\text{seq}} \), the aggregate transition net gain vector \( (NG)^{k+v}_k \) is given by the summation \( \sum_{i=k}^{k+v} NG^{(i)} \).

Lemma 1. Let \( \Pi \) be an SN P system with \( m \) neurons and \( m \) rules (one rule for each neuron), \( M_{\Pi} \) be the spiking transition matrix of \( \Pi \), and \( (SV)^{k+v}_k \) be the summation of a finite consecutive subsequence of \( SV_{\text{seq}} \), with the elements \( (s^{(k)}, s^{(k+1)}, \ldots, s^{(k+v)}) \) representing the series of spiking vectors during the \( k \)th, \( (k+1) \)th, \( \ldots \), \( (k+v) \)th time-steps, respectively. Then the total net gain of spikes for neurons \( 1 \ldots m \) during the span of \( k \)th \( \ldots \), \( (k+v) \)th time-steps is equal to the aggregate transition net gain vector \( (NG)^{k+v}_k \), computed using the formula:

\[
(NG)^{k+v}_k = (SV)^{k+v}_k \cdot M_{\Pi}
\]  

(4)

Proof. From Equation 2, we know the following:

\[
NG^{(k)} = s^{(k)} \cdot M_{\Pi}
\]  

(5)

\[
NG^{(k+1)} = s^{(k+1)} \cdot M_{\Pi}
\]  

(6)
\[ NG^{(k+v)} = s^{(k+v)} \cdot M_{\Pi} \] (7)

Summing the left and right sides of these equations, we get the following:

\[ NG^{(k)} + NG^{(k+1)} + \ldots + NG^{(k+v)} = s^{(k)} \cdot M_{\Pi} + s^{(k+1)} \cdot M_{\Pi} + \ldots + s^{(k+v)} \cdot M_{\Pi} \] (8)

Factoring out \( M_{\Pi} \) from the right side of Equation 8, and knowing that the summation \( NG^{(k)} + NG^{(k+1)} + \ldots + NG^{(k+v)} \) is exactly the aggregate transition net gain vector \( (NG)^{k+m}_{k} \) from Definition 11, we get the following:

\[ (NG)^{k+v}_{k} = (s^{(k)} + s^{(k+1)} + \ldots + s^{(k+v)}) \cdot M_{\Pi} \] (9)

But the summation \( (s^{(k)} + s^{(k+1)} + \ldots + s^{(k+v)}) \) is exactly the aggregate spiking vector \( (SV)^{k+v}_{k} \) from Definition 9. Thus,

\[ (NG)^{k+v}_{k} = (SV)^{k+v}_{k} \cdot M_{\Pi} \] (10)

Lemma 1 generalized Equation 2, to describe the net gain of spikes in neurons after a series of computation steps. The following lemma likewise generalizes Equation 1.

**Lemma 2.** Let \( \Pi \) be an SN P system with \( m \) neurons and \( m \) rules (one rule for each neuron), and \( C_{k} \) be the configuration of the system at time-step \( k \). After \( v \) time-steps, the configuration \( C_{k+v} \) is obtained by the following:

\[ C_{k+v} = C_{k} + (NG)^{k+v}_{k} \] (11)

**Proof.** This can be easily derived using the same line of reasoning in proving Lemma 1.

\[ \square \]

The following two more definitions will be crucial in formulating our main theorem on periodicity in SN P systems.
Definition 12 (Zero-Aggregate Transition Net Gain Vector). Let Π be an SN P system with m neurons and m rules (one rule for each neuron), and NG seq = (NG(0), NG(1), NG(2), ...) be the transition net gain vector sequence of Π. If any aggregate transition net gain vector (NG)k+v sums up to zero, represented by the vector (0, 0, 0, ..., 0), we call it the zero-aggregate transition net gain vector.

Definition 13 (Null Aggregate Spiking Vector). Let Π be an SN P system with m neurons and m rules (one rule for each neuron), SV seq = (s(0), s(1), s(2), ...) be the spiking vector sequence, and (SV)k+v be the aggregate spiking vector during the kth, k+1th, ..., (k+v)th computation steps of Π. If the aggregate transition net gain vector (NG)k+v, as computed by the formula (NG)k+v = (SV)k+v · MΠ (Equation 10 from Lemma 1) is equal to zero (a zero-aggregate transition net gain vector), then we call (SV)k+v a null aggregate spiking vector.

4.2 A Necessary Condition for Periodicity in SN P Systems

First, we claim that the Null Aggregate Spiking Vector and its resulting Zero-Aggregate Transition Net Gain Vector in definitions 13 and 12, respectively, corresponds to a deterministic, periodic SN P system.

Lemma 3. Let Π be an SN P system with m neurons and m rules (one rule for each neuron), MΠ be the spiking transition matrix, and SV seq be the spiking vector sequence of Π. The system is periodic if and only if there exist a Null Aggregate Spiking Vector (SV)k+v from a finite consecutive subsequence of SV seq which results to a Zero-Aggregate Transition Net Gain Vector (i.e., (NG)k+v = 0).

Proof. Let Ck be the configuration of the system Π, and s(k) be the spiking vector during time step k. We also let s(k) be the first element of the finite consecutive subsequence summed up in (SV)k+v. If the aggregate spiking vector (SV)k+v during the time-steps k, k+1, ..., k+v is a Null Aggregate Spiking Vector, resulting to a Zero-Aggregate Transition Net Gain Vector (i.e., (NG)k+v = 0), then the configuration Ck+v, using Equation 11, is obtained as Ck+v = Ck + (NG)k+v = Ck + 0 = Ck. This means that the configuration Ck during time-step k is repeated during time-step (k + v). Thus, by Definition 7, the system Π is periodic.

Moreover, given a periodic system Π, then it implies that at least one configuration, say Ck, is repeated after a particular amount of time-steps, say v, where Ck = Ck+v. This means that (NG)k+v = Ck+v - Ck = 0. Thus there must exist a Null Aggregate Spiking Vector (SV)k+v that yielded the Zero-Aggregate Transition Net Gain Vector (NG)k+v = 0).

The following theorem presents a necessary condition for SN P systems to be periodic. We claim that if an SN P system Π is periodic, then the determinant of its spiking transition matrix MΠ must be equal to zero.
Theorem 1. Let $\Pi$ be an SN P system with $m$ neurons and $m$ rules (one rule for each neuron), and $M_\Pi$ be the spiking transition matrix of $\Pi$. If $\Pi$ is periodic, then the determinant of the $m \times m$ matrix $M_\Pi$ must be equal to zero.

Proof. If $\Pi$ is periodic, then from Lemma 3, there must exist a Null Aggregate Spiking Vector $(SV)^{k+v}_k$ that yields a Zero-Aggregate Transition Net Gain Vector $((NG)^{k+v}_k = 0)$. Recall Equation 4 from Lemma 1:

\[(NG)^{k+v}_k = (SV)^{k+v}_k \cdot M_\Pi\] (12)

Since $(NG)^{k+v}_k = 0$, then,

\[(SV)^{k+v}_k \cdot M_\Pi = 0\] (13)

Solving for the Null Aggregate Spiking Vector $(SV)^{k+v}_k$ in Equation 13 is equivalent to finding the null space of the matrix $M_\Pi$. We know that this equation has at least one solution: the zero vector $(0^{(1)}, 0^{(2)}, ..., 0^{(m)})$ [2]. But we ignore such a solution here because a spiking vector with all components zero implies that the system has not spiked (i.e., no rule has been applied.) Furthermore, we know that this equation will have solutions other than the trivial solution if and only if the matrix $M_\Pi$ is singular [2]. For square matrices (which we know is true for the $m \times m$ matrix $M_\Pi$), if the matrix is singular, then the determinant of the matrix $M_\Pi$ must be equal to zero. Thus if $\Pi$ is periodic (implying that there exist a Null Aggregate Spiking Vector $(SV)^{k+v}_k$), then Equation 13 must have solutions other than the trivial one, which means that the determinant of the matrix $M_\Pi$ must be equal to zero.

\[\square\]

5 Examples and Empirical Verifications

In the previous section, we restricted our definitions and theorems to SN P systems with exactly one rule per neuron, which makes them deterministic. At first glance, this restriction might appear very limiting, as these types of systems are rarely used in the generative mode, since non-determinism is what gives the system flexibility to generate multiple outputs from the same initial configuration. We justify this restriction in the previous section by claiming that a 1-rule-per-neuron deterministic periodic system is the core skeleton of the structure of a nondeterministic system that is finite (i.e., consists of finite number of neurons) but able to generate an infinite number of outputs.

In this section, we empirically verify such claim by examining some examples of finite generative SN P systems that generate an infinite number of outputs. These systems are taken from examples in [4]. We present three particular systems, in the following process: (1) each example starts off with a description of how the system runs and how it is able to generate its output, (2) we then proceed to eliminate some parts of the system, namely the output neuron and
some rules (usually the forgetting rules and the ones with delay), leaving us a de-
terministic, periodic, 1-rule-per-neuron SN P system, and (3) lastly, we present
the spiking transition matrices of these systems, noting that their determinants
are all zero (verifying Theorem 1).

Example 1 - an SN P system generating all even natural numbers, from [4].

Figure 1 shows the graphical representation of the system. We refer the
reader to [4] for the more detailed discussion on the system.

In this and in the next examples, the output of the generative SN P System
is the difference between the first two time-steps during which the designated
output neuron fires. After the second firing of the output neuron, no rules are
applicable and the system halts.

The SN P System in Figure 1 works in the following way. Initially, only
neurons 1, 2, 3, and 7 (the output neuron) contain spikes, and they all fire in
the first step. The firings of neurons 1, 2, and 3 send a spike each to neurons
4, 5, and 6, respectively. In step 2, the roles are reversed - neurons 4, 5, and 6
fire, sending spikes back to the first 3 neurons. This pattern repeats indefinitely,
breaking the cycle only when the rule \((a \rightarrow a; 1)\) in neuron 4 is applied instead
of \((a \rightarrow a; 0)\), subsequently enabling the output neuron 7 to fire again, thus
producing the output. We note that the period of the pattern described above
is 2, enabling the system to generate outputs starting from the number 2, with
increments of 2, thus generating all the even natural numbers.
In the context of our discussions in Section 4, we isolate particular neurons and rules in the SN P System in Figure 1, parsing out a deterministic, one-rule-per-neuron system. Specifically, we remove the output neuron (neuron 7), the rule with 1 time-step delay in neuron 4, and the forgetting rules of neurons 1, 2, and 3. What we obtain is the system in Figure 2.

![Figure 2](image)

**Fig. 2.** A deterministic, periodic SN P system with period = 2.

The spiking transition matrix $M_1$ of this 6-neuron, 6-rule SN P System (assigning the rules with the labels of their respective neurons), is a $6 \times 6$ matrix shown in the following:

$$M_1 = \begin{bmatrix}
-2 & 0 & 0 & 1 & 0 & 0 \\
0 & -2 & 0 & 0 & 1 & 0 \\
0 & 0 & -2 & 0 & 0 & 1 \\
1 & 1 & 1 & -1 & 0 & 0 \\
1 & 1 & 0 & 0 & -1 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 
\end{bmatrix}$$

We note that the determinant of the matrix $M_1$ shown above is equal to zero, thus empirically verifying our claim in Theorem 1 that periodicity of a 1-rule-per-neuron SN P system implies that the determinant of its spiking transition matrix is zero.
Example 2 - an SN P system generating all natural numbers greater than 1, from [4]. Figure 3 shows the graphical representation of the system. We refer the reader to [4] for the more detailed discussion on the system. The system in Figure 3 works as follows. At the onset, all neurons contain the required number of spikes, enabling them to fire. These firings send a new spike each to neurons 1 and 2, while sending two spikes to neuron 3. In the next time-step, neurons 1 and 2 can fire again, whereas the two spikes received by neuron 3 is not enough for it to be able to fire. Subsequent firings of neurons 1 and 2 results to a periodic pattern, whose period is 1. The cycle is broken only when the rule \((a \rightarrow a; 1)\) in neuron 2 is applied instead of the rule \((a \rightarrow a; 0)\), resulting to the output neuron (neuron 3) firing again, and the system halts. In the same sense that the system in the first example, with period = 2, produced a set of outputs with interval 2, the system in the example here, with period = 1, produced a set of outputs with interval 1.

Likewise, we isolate what we deem the essential components of the SN P System in Figure 3. Specifically, we remove the output neuron (neuron 3), the forgetting rule in neuron 1, and the rule with a 1 time-step delay in neuron 2. We obtain the system in Figure 4. The spiking transition matrix \(M_2\) of this system is likewise shown below:

\[
M_2 = \begin{bmatrix}
-1 & 1 \\
1 & -1
\end{bmatrix}
\]

We note that the determinant of the spiking transition matrix \(M_2\) shown above is equal to zero, again empirically verifying our main theorem.
Fig. 4. A deterministic, periodic SN P system, with period = 1.

Example 3 - an SN P system generating all natural numbers, from [4]. Figure 5 shows the graphical representation of the system. We refer the reader to [4] for the more detailed discussion on the system.

Fig. 5. An SN P system generating all natural numbers, from [4].

The system in Figure 5 works in a very similar way as the system in Figure 3, which is expected since the only difference in the set that they generate is a
single element: the number 1. We likewise note that the period of the pattern in this system is also equal to 1.

We proceed with eliminating some components of this system. We remove the output neuron (neuron 4), the forgetting rules in neurons 1 and 2, and the rule with a 1 time-step delay in neuron 4. We obtain the system shown in Figure 6.

![Fig. 6. A deterministic, periodic SN P system, with period = 1.](image)

The spiking transition matrix $M_3$ of this system is shown below:

$$M_3 = \begin{bmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{bmatrix}$$

As with the first two examples, the determinant of the spiking transition matrix $M_3$ is equal to zero.

We have thus presented some examples of SN P systems that generate an infinite number of outputs (particularly, numbers of some interval), and empirically verified our claim that within the core structure of these systems is a deterministic, periodic, 1-rule-per-neuron SN P system. We obtain such a deterministic, periodic, 1-rule-per-neuron SN P system from the from each of the
examples (which are all nondeterministic, more-than-1-rule-per-neuron systems) by following the 3-step process described at the start of this section.

Moreover, if we reverse the process described above, that is: (1) start with a spiking transition matrix (must be a square matrix) with determinant equal to zero, and create the deterministic, periodic, 1-rule-per-neuron SN P system corresponding to such spiking transition matrix; (2) add some elements to the system - i.e., an output neuron and some additional rules on some neurons (usually rules with delay); and (3) we then have a nondeterministic, finite SN P system that is able to generate an infinite number of outputs. The reversed process just described serves as a framework for the design of generative SN P Systems.

Note that step (1) in the reversed process just described above, where a deterministic, periodic, 1-rule-per-neuron SN P system is created from the spiking transition matrix, is not a straightforward task. This is precisely because it presumes that a spiking transition matrix with determinant equal to zero is a sufficient condition for obtaining a periodic, 1-rule-per-neuron SN P system, whereas Theorem 1 provides only a necessary condition. This is subject to an appropriate choice of the initial configuration $C_0$. Recall that the spiking transition matrix only stores information on the effect of each of the rules to each of the neurons, and does not include information on the system’s configuration (number of spikes in each neuron). By appropriately choosing an initial configuration such that some time in the computation sequence of the system, periodicity occurs, then the task in step (1) above is achieved. We thus present this as an open problem: formulating a sufficient condition for periodicity in SN P systems, subject to additional assumptions or restrictions.

Lastly, we emphasize that we deem that periodicity is an essential and inherent aspect of generative SN P systems, and that their ability to generate an infinite number of outputs, given only a finite number of neurons and rules, lies on them being periodic.

6 Final Remarks

In this paper, we explored the study of periodicity in SN P systems as one of its dynamical aspects. We used matrix representation and other algebraic concepts, together with some of our additional definitions, to describe periodic properties in an SN P System.

We adapted some examples from the literature of SN P systems and successfully showed some empirical verification of our claims.

We believe that these algebraic representations, aside from being an elegant representation from a theoretical point of view, will also be instrumental in providing some sort of a design framework for SN P systems, as described in the last part of the previous section.
Acknowledgments

The authors wish to acknowledge the support of the ERDT program of DOST-SEI, and express appreciation towards the Algorithms and Complexity Laboratory. They likewise are thankful for the encouragement, discussions and insights of their colleagues in the said laboratory, especially to Mr. Francis Cabarle.

References

Formal Verification of P Systems with Active Membranes through Model Checking

Florentin Ipate¹, Raluca Lefticaru¹, Ignacio Pérez-Hurtado², Mario J. Pérez-Jiménez², and Cristina Tudose¹

¹Department of Computer Science, University of Pitesti
Str. Targu din Vale 1, 110040, Pitesti, Romania
{florentin.ipate,raluca.lefticaru,cristina.tudose}@upit.ro

²Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012, Sevilla, Spain
{perezh,marper}@us.es

Abstract. Formal verification of P systems using model checking has attracted a significant amount of research in recent years. However, up to now only P systems with static structure have been considered. This paper makes significant advances in this area by considering P systems with active membranes, in particular P systems with division rules. The paper presents a theoretical framework for addressing this problem and reports on a complex case study involving a well-known NP-complete problem solved using P systems with membrane division rules. This is implemented in Promela and non trivial properties are verified using Spin.

1 Introduction

Inspired by the behaviour and functioning of a living cell, membrane computing has emerged in recent years as a powerful modelling tool; various applications have been reported [4], especially in biology and bio-medicine, but also in many other areas, such as economics, approximate optimization and computer graphics [20]. Furthermore, software tools, such as P-Lingua [14], for simulating P systems, have been developed and used in real life problems. Naturally, such modelling and simulation tools must be accompanied by appropriate means of formally verifying that the model satisfies the required properties.

One of the most widely used approaches to formal verification is through model checking. This uses a model of the implementation, given as an operational specification, and a specification, given as a temporal logic formula, and verifies, on the entire state space of the model, whether the property holds or not. If the property fails, then a counterexample is also returned.

Formal verification of P systems using model-checking has attracted a significant amount of research in recent years, using tools such as Maude [1], PRISM [21], NuSMV [13], Spin [7], [12] or ProB [10]. However, up to now only P systems
with static structure have been considered. Some of the aforementioned investigations consider cell dissolution rules, but in this case a simple flag, indicating whether the membrane exists or not in the current configuration, can be used to address the change in the membrane structure. This is, indeed, a clear weakness of these approaches since, in biology, the membrane structure is not static, but it evolves and changes in time. Furthermore, P systems with active membranes have a wide range of applications; in particular, P systems with division rules are used to devise efficient solutions to computationally hard problems [20].

This paper makes significant advances in the area of model checking based verification of P systems by considering P systems with active membranes. Firstly, it devises a theoretical framework for addressing this problem: it describes the Kripke structure associated with a P system with active membranes and how this can be translated into an executable implementation; it also shows how properties that can be formulated for the P system can be translated into properties of the executable implementation. Secondly, it reports on a complex case study involving a well-known NP-complete problem solved in linear time and in an uniform way using P systems with membrane division rules [17]. The solution was improved in [8], where the total cost is logarithmic in one variable and linear in the rest, but this improvement is not essential in the context of this paper. For this example, a number of non-trivial properties of the model are formulated and verified using the Spin model checker [2].

The paper is structured as follows. We start by presenting in Section 2 the notation and main concepts to be used in the paper. Section 3 presents the theoretical background for our approach. The case study and empirical results are given in the next two sections, while Section 6 discusses related work. Finally, conclusions are drawn in Section 7.

2 Background

2.1 P Systems

Before presenting our approach to P system verification, let us establish the notation used and define the class of cell-like P systems addressed in the paper. Basically, a P system is defined as a hierarchical arrangement of membranes, identifying corresponding regions of the system. Each region has an associated finite multiset of objects and a finite set of rules; both may be empty. Given a finite alphabet \( V = \{ a_1, \ldots, a_p \} \), a multiset is either denoted by a string \( u \in V^* \) (in which the order is not important, the string notation is only used as a convention), or by an associated vector of non-negative integers, \( \Psi_V(u) = (|u|_{a_1}, \ldots, |u|_{a_p}) \), where \(|u|_{a_i}\) denotes the number of \( a_i \) occurrences in \( u \), for each \( 1 \leq i \leq p \).

The following definition refers to cell-like P systems with active membranes (see [19] for details).

**Definition 1.** A P system is a tuple \( H = (V, H, \mu, w_1, \ldots, w_n, R) \), where \( V \) is a finite set, called alphabet; \( H \) is a finite set of labels for membranes; \( \mu \) is a
Formal Verification of P Systems with Active Membranes.

membrane structure (a rooted tree), consisting of \( n \) membranes injectively labelled with elements of \( H; w_i, 1 \leq i \leq n \), ... configurations; \( c \) is a halting configuration if there is no region \( i \) such that its contents can be further developed.

(a) \([a \rightarrow v]_h^\alpha\), where \( h \in H, \alpha \in \{+,-,0\} \) (electrical charges), \( a \in V \) and \( v \) is a string over \( V \) describing a multiset of objects associated with membranes and depending on the label and the charge of the membranes (evolution rules).

(b) \([a]_h^\alpha \rightarrow [b]_h^\beta\), where \( h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in V \) (send-in communication rules). An object is introduced in the membrane, possibly modified, and the initial charge \( \alpha \) is changed to \( \beta \).

(c) \([a]_h^\alpha \rightarrow [b]_h^\beta\), where \( h \in H, \alpha, \beta \in \{+,-,0\}, a, b \in V \) (send-out communication rules). An object is sent out of the membrane, possibly modified, and the initial charge \( \alpha \) is changed to \( \beta \).

(d) \([a]_h^\alpha \rightarrow b\), where \( h \in H, \alpha \in \{+,-,0\}, a, b \in V \) (dissolution rules). A membrane with a specific charge is dissolved in reaction with an object \( a \) (possibly modified).

(e) \([a]_h^\alpha \rightarrow [b]_h^\beta[c]_h^\gamma\), where \( h \in H, \alpha, \beta, \gamma \in \{+,-,0\}, a, b, c \in V \) (division rules). A membrane is divided into two membranes. The objects inside the membrane are replicated, except for \( a \), that may be modified in each membrane.

The membrane structure, \( \mu \), is denoted by a string of left and right brackets \(([\mu]_l^l \text{ and } [\mu]_r^r)\), each with the label \( l \) of the membrane; the electrical charge \( e \) of each membrane is also given. The environment is only used to send the answer to and we do not capture its structure into this definition.

The rules are applied in maximally parallel mode, which means that they are used in all the regions at the same time and in each region all the objects to which a rule can be applied must be the subject of a rule application [19]. However, any membrane can be subject of only one rule of types (b) – (e) in one computation step. In type (e) (membrane division) rules, all the contents present before the division, except for object \( a \), can be the subject of rules in parallel with the division. In this case we consider that in a single step two processes take place: first the contents are affected by the rules applied to them, and after that the results are replicated into the two new membranes. If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external membrane which has not been dissolved at that computation step. The skin is never dissolved neither divided. The behaviour of this system is precisely the same as the behaviour mentioned in [15–17].

A configuration of the P system \( \Pi \) is uniquely identified by the current membrane structure \( \mu' \) and the contents of each region in \( \mu' \). A transition step from a configuration \( c_1 \) to a configuration \( c_2 \) is realized if the P system can evolve from \( c_1 \) to \( c_2 \) by using the maximal parallelism mode (as well as the rule restrictions stated above); this is denoted by \( c_1 \Rightarrow c_2 \). In the set of all configurations, we will distinguish halting configurations; \( c \) is a halting configuration if there is no region \( i \) such that its contents can be further developed.
2.2 Linear Temporal Logic

The Linear Temporal Logic (LTL) was introduced by Amir Pnueli in 1977 [18] for the verification of computer programs. Compared to CTL (Computation Tree Logic) [5], LTL does not have an existential path quantifier (the E of CTL). An LTL formula has to be true over all paths, having the form $A f$, where $f$ is a path formula in which the only state subformulas permitted are atomic propositions.

Given a set of atomic propositions $AP$, an LTL path formula [5] is either:

- If $p \in AP$, then $p$ is a path formula.
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $f \land g$, $X f$, $F f$, $G f$, $f U g$ and $f R g$ are path formulas, where:
  - The $X$ operator (“neXt time”, also written $\bigcirc$) requires that a property holds in the next state of the path.
  - The $F$ operator (“eventually” or “in the future”, also written $\lozenge$) is used to assert that a property will hold at some state on the path.
  - $G f$ (“always” or “globally”, also written $\square$) specifies that a property, $f$, holds at every state on the path.
  - $f U g$ operator (U means “until”) holds if there is a state on the path where $g$ holds, and at every preceding state on the path, $f$ holds. This operator requires that $f$ has to hold at least until $g$, which holds at the current or a future position.
  - $R$ (“release”) is the logical dual of the $U$ operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually: if $f$ never becomes true, $g$ must remain true forever.

3 Theoretical Basis for Model Checking of P Systems with Active Membranes

In this section we describe the transformation of a P system with active membranes into a Kripke structure, by extending the approach given in [7] and [11] for P systems with static structure.

Definition 2. A Kripke structure over a set of atomic propositions $AP$ is a four tuple $M = (S, H, I, L)$, where $S$ is a finite set of states; $H \subseteq S \times S$ is a transition relation that must be left-total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in H$; $I \subseteq S$ is a set of initial states; $L : S \rightarrow 2^{AP}$ is an interpretation function, that labels each state with the set of atomic propositions true in that state.

Usually, the Kripke structure representation of a system results by giving values to every variable in each configuration of the system. Suppose $var_1, \ldots, var_k$ are the system variables, $Val_i$ denotes the set of values for $var_i$, and $val_i$ is a value from $Val_i$, $1 \leq i \leq k$. Then the states of the system are $S = \{ (val_1, \ldots, val_k) | val_1 \in Val_1, \ldots, val_k \in Val_k \}$, and the set of atomic predicates are $AP =$
\{(\text{var}_i = \text{val}_i) \mid 1 \leq i \leq k, \text{val}_i \in \text{Val}_i\}. Naturally, \(L\) will map each state (given by the values of variables) onto the corresponding set of atomic propositions. For convenience, in the sequel the expressions of \(AP\) and \(L\) will not be explicitly given, the implication being that they are defined as above.

Obviously, in a P system with division rules the number of membranes can grow infinitely. For practical reasons, we will allow only a finite number of membranes and will assume that the upper bound \(k\) on this number is known beforehand. However, this is not a limitation because, given a specific P system, this upper bound can be estimated. Then, a configuration of a P system can be represented by a tuple \(c = (\text{var}_1, \ldots, \text{var}_k)\), where \(\text{var}_i\) holds the current contents \(u_i\) of region \(i\) (a special symbol not in \(V\) will be used when the \(i\)th membrane does not exist), the membrane label, its electrical charge and the number of the parent membrane.

Consider a P system \(\Pi = (V, H, \mu, w_1, \ldots, w_n, R)\) with \(R = \{r_1, \ldots, r_m\}\). The states of the Kripke structure associated with \(\Pi\) will correspond to the configurations of the P system (plus two special states, as explained later). Given two configurations, \(c\) and \(d\), there is a transition from \(c\) to \(d\) if for every membrane \(i\) in \(c\) there exist \(n_1^i, \ldots, n_m^i\), such that the following conditions hold simultaneously:

- at least one rule is applied, i.e. there exists \(i\) such that \(n_1^i + \ldots + n_m^i > 0\);
- \(d\) is obtained from \(c\) by applying rules \(r_1, \ldots, r_m\), \(n_1^i, \ldots, n_m^i\) times, respectively, for every membrane \(i\);
- any membrane can be subject of only one rule of type (b)–(e), i.e. for every \(i\), there exists at most one \(j\), \(1 \leq j \leq m\), such that \(n_j^i > 0\) and rule \(r_j\) is of type (b)–(e); furthermore, in this case \(n_j^i = 1\);
- a computation from \(c\) develops in maximally parallel mode, i.e. for every membrane \(i\) and every \(j\), \(1 \leq j \leq m\), if rules \(r_1, \ldots, r_m\) can be applied \(n_1^i, \ldots, n_j^i - 1, (n_j^i + 1), n_{j+1}^i, \ldots, n_m^i\) times, respectively, in \(c\) then: (1) \(r_j\) is of type (b)–(e) and (2) there exists \(l\), \(1 \leq l \leq m\) such that \(n_l\) is of type (b)–(e) and \(n_l^i = 1\).

In order to keep the number of states finite, for each configuration \(u\) we will assume that each component of \(\Psi_V(u)\) has an established upper bound, denoted \(\text{Max}\), and each rule can only be applied for at most a given number of times, denoted \(\text{Sup}\) (therefore \(n_1^1, \ldots, n_m^m \leq \text{Sup}\) in the above definition). Whenever (at least) one of these upper bounds is exceeded, extra transitions to a special state, \(\text{Crash}\), are added. Consequently, as long as the verification is performed for “specific” P systems (and not for a class of P systems), then this upper bound is not a limiting factor. However, the upper bounds and the \(\text{Crash}\) state are added just to prevent an unexpected behaviour, caused by a P system with a misleading computation.

The halting configurations of the P system (i.e. in which no rule can be applied) are also represented by extra transitions, to another special state, \(\text{Halt}\). In order for the transition relation to be left-total, loop-back transitions in \(\text{Crash}\) and \(\text{Halt}\) are also added.

One additional problem arises when implementing the Kripke structure defined above: most modelling languages (Promela included) do not support the
Table 1. Reformulating the basic LTL operators for the Promela specification of a P system

<table>
<thead>
<tr>
<th>Property</th>
<th>LTL specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>G p</td>
<td>$[\cdot (p||pInS)]$</td>
</tr>
<tr>
<td>F p</td>
<td>$&lt;&gt; (p&amp;&amp;pInS)$</td>
</tr>
<tr>
<td>p U q</td>
<td>$(p||pInS) U (q&amp;&amp;pInS)$</td>
</tr>
<tr>
<td>X p</td>
<td>$X ({pInS U (p&amp;&amp;pInS)})$</td>
</tr>
<tr>
<td>p R q</td>
<td>$(p&amp;&amp;pInS) V (q||pInS)$</td>
</tr>
</tbody>
</table>

existential (or the universal) quantifier. Consequently, a transition involving $\exists$ (as in the Kripke structure representation of a P system) is normally implemented as a sequence of transitions (e.g. a “do - od” loop in Promela) and so additional (intermediary) states are introduced into the model. Naturally, the intermediary states cannot be allowed to form infinite loops and so every possible path in the Promela executable model will contain infinitely often states corresponding to the P system configurations. These assumptions ensure that every path in the P system has at least one corresponding path in the Promela model and vice versa.

Naturally, the properties to be verified (and which refer to the given P system) will need to be reformulated as equivalent formulas for the associated Promela model. Table 1 summarizes the transformations of basic LTL formulas for the Promela implementation, as formally proven in [12] ($pInS$ is a predicate which holds in the original (non-intermediary) states). For example, ‘always $b > 0$’ (the number of occurrences of $b$ objects is always greater than 0), will become, for the Promela model, ‘Globally $b > 0$ or not $pInS$’ (we expect $b > 0$ only for configurations corresponding to the P system, but not for the intermediary states).

4 Case Study: The Subset Sum Problem

The Subset Sum problem has been used with some classes of P systems with active membranes in order to illustrate the efficiency and power of these mechanisms for solving NP-complete problems in an uniform way ([15, 17, 9, 16, 8]). Most of these solutions follow a similar scenario with a number of stages: generation, calculation, checking and output. This is important for the approach we suggest as we can identify specific properties for some of these stages as well as global ones.

The Subset Sum problem can be formulated as follows:

*Given a finite set $A = \{a_1, \ldots, a_n\}$, of $n$ elements, where each element $a_i$ has an associated weight, $w_i$, and a constant $k \in \mathbb{N}$, it is requested to determine whether or not there exists a subset $B \subseteq A$ such that $w(B) = k$, where $w(B) = \sum_{a_i \in B} w_i$.**
Formal Verification of P Systems with Active Membranes...

For given $n$ and $k$, a P system with active membranes $\Pi((n, k))$ is constructed to solve the Subset Sum problem - for details see [17]. The P system is given by

$$\Pi((n, k)) = (\Gamma((n, k)), \{e, s\}, \mu, w_e, w_s, R, i(n, k)),$$

where

- $\Gamma((n, k)) = \{x_0, x_1, \ldots, x_n\} \cup \{a_0, a, a_0, a_1, a_0, \ldots, a_n, q, q_0, \ldots, q_{2k+1},$
- $z_0, \ldots, z_{2n+2k+2}, \text{Yes}, N_0, \text{No}, \#\}$ is the alphabet;
- $\mu = \{\mu_i\}_{i=1}^n$ is the membrane structure;
- $w_e = z_0, w_s = e_0a^k$ are the initial multisets;
- $i(n, k) = e$ and contains the code $x_1^{w_1} \ldots x_n^{w_n}$;
- the set of evolution rules, $R$, consists of

1. $[e_1]_e^0 \rightarrow [q]_e^1 [e_1]_e^+, 0 \leq i \leq n$;
2. $[e_i]_e^+ \rightarrow [e_{i+1}]_e^0 [e_i]_e^+, 0 \leq i \leq n - 1$. For each subset of $A$ a membrane is generated.
3. $[x_0 \rightarrow a_0]_e^0, [x_0 \rightarrow \lambda]_e^+, [x_i \rightarrow x_{i-1}]_e^+$, for $1 \leq i \leq n$.
   The code from the input membrane is built in such a way that the multiplicity of $x_j$ represents the weight of $a_j \in A$. These three rules calculates in $a_0$ the weight of a subset.
4. $[q \rightarrow q_0]_e^0; [a_0 \rightarrow a]_e^0; [a \rightarrow a]_e^0$.
   These rules mark the beginning of the checking stage; the weight of the subset is now coded by the multiplicity of $a_0$.
5. $[a_0]_e^0 \rightarrow [\#]_e^0; [a]_e^0 \rightarrow [\#]_e^0$.
   The number of occurrences of $a_0$ and $a$ are compared in a checking loop.
6. $[q_j \rightarrow q_{j+1}]_e^0, 0 \leq j \leq k; [q_{j+1} \rightarrow q_{j+2}]_e^0, 0 \leq j \leq k - 1$.
   Objects $q_i$ are utilised as counters of the checking loop.
7. $[d_j]_e^0 \rightarrow [\#]_e^0 d_j; [N_0]_e^0 \rightarrow [\#]_e^0 N_0$.
   These rules provide an answer to the checking loop given that there are the same number of $a_0$ and $a$, more $a_0$ objects, or more $a$ objects, respectively.

5 Experimental Results

In the sequel we will show how different properties of the above P system, providing an answer to an instance of the Subset Sum problem, can be checked. We start with some simpler properties and then will focus on some more complex ones that have been revealed by the current literature [17]. Before starting the presentation of these properties it is important to mention that many of the properties require some transformations or adaptations in order to be checked and validated - as discussed in Section 3.

We have considered different instances of the Subset Sum problem. We have considered a set $A$ with two and three elements, weights between one and three,
Table 2. Some properties verified for the Promela specifications of two P systems, providing an answer to the Subset Sum problem.

<table>
<thead>
<tr>
<th>Property</th>
<th>LTL specification</th>
<th>Result 1</th>
<th>Result 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generally, there is not YES in the environment.</td>
<td>![env.Yes == 0</td>
<td></td>
<td>!pInS](expression)</td>
</tr>
<tr>
<td>Generally, there is not NO in the environment.</td>
<td>![env.No == 0</td>
<td></td>
<td>!pInS](expression)</td>
</tr>
<tr>
<td>Eventually, there is YES in the environment.</td>
<td>&lt;&gt;(!env.Yes == 1 &amp; &amp; pInS)</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Eventually, there is NO in the environment.</td>
<td>&lt;&gt;(!env.No == 1 &amp; &amp; pInS)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>When $e_1$ appears in a specific membrane 1, with electrical charge 0, then a $q$ will appear with a negative electrical charge.</td>
<td>![ memb[1].e[1]==1 &amp; &amp; memb[1].charge==0 -&gt; &lt;&gt;( memb[1].q==1 &amp; &amp; memb[1].charge==-1))</td>
<td></td>
<td>pInS](expression)</td>
</tr>
<tr>
<td>When $e_1$ appears in a specific membrane 1, with electrical charge 0, then a $q$ will appear with a positive electrical charge.</td>
<td>![ memb[1].e[1]==1 &amp; &amp; memb[1].charge==0 -&gt; &lt;&gt;( memb[1].q==1 &amp; &amp; memb[1].charge==1))</td>
<td></td>
<td>pInS](expression)</td>
</tr>
<tr>
<td>For all $i, j$, if $e_i$ appears in membrane $j$, with electrical charge 0, then $q$ will appear in the same membrane with a negative electrical charge.</td>
<td>![ (i&gt;=0 &amp; &amp; i&lt;3 &amp; &amp; j&gt;=0 &amp; &amp; j&lt;10 &amp; &amp; ( memb[j].e[i]==1 &amp; &amp; memb[j].charge==0 -&gt; &lt;&gt; ( memb[j].q==1 &amp; &amp; memb[j].charge==-1 &amp; &amp; pInS)))</td>
<td></td>
<td>pInS](expression)</td>
</tr>
</tbody>
</table>

and values of $k$ between two and four. We have tested both cases, with and without solution. In Table 2 we present some properties verified for the Promela specification of the P system. The first two columns express the properties to be verified and the last two columns the results obtained for two instances of the Subset Sum P systems. The first example is $n = 3, k = 4, w = [1, 2, 2]$ and has the associated results in the third column, while the second example is $n = 3, k = 3, w = [2, 2, 2]$ and the model checker answers are given in the last column of Table 2.

We first checked whether there is a solution for a given instance of Subset Sum (“Generally, there is not NO in the environment”). As said above we had to slightly change the query due to the current codification of the P system into Promela and the associated Kripke structure. The current form of the query is ![env.No == 0 || pInS](expression), which means that we consider only states associated with the P system and not intermediary ones and check the variable No from
the environment, env.No, to be 0. The same behaviour of the system can be checked by “Eventually, there is YES in the environment”.

Based on rule of the type “(1)” it can be shown that when $e_i$ appears in a specific $e$ membrane, or more generally, in any such membrane, with electrical charge 0, then in the membrane obtained from it with negative electrical charge, $q$ will appear.

We have also made some experiments running simulations with both P-Lingua and the Promela code. The results obtained were the same (final configuration, number of steps, obtained membranes etc.) and the two simulations have produced values of the same order of magnitude regarding the elapsed time. This shows that the Kripke structure underlying the translation from P-Lingua to Promela does not introduce much overhead into the system.

For the two instances of the Subset Sum P systems considered before, the maximum number of membranes obtained during the P system computation was 16 (corresponding to $n = 3$) and the verification time of each property listed in Table 2 was 7 – 8 seconds.

In future experiments we will test the limit for which Spin can produce a response in a reasonable time interval and will investigate how our implementation can be improved in order to cope with the well-known state explosion problem associated with model checking. We also aim to address more complex properties as well as to identify invariants of various stages.

6 Related Work

A first approach on P system verification using model checking is introduced in [1]. The authors transform P systems into executable specifications written in Maude, a programming language and the software system developed around it, that uses a rewriting semantics. Further, the LTL model checker of Maude is used to verify some properties of the P systems.

Other papers [6, 7] tackle the decidability problem of P system properties. The models used for experiments are, similarly to those in [1], simple cell-like P systems, generating for example $n^2$. They present membrane dissolving rules, but no membrane division. The tools used for experimentation are Omega and Spin, the authors conclusion is that Spin is preferable over Omega for model checking P systems.

Another paper that compares the experimental results obtained by two main stream model checkers is [12], which concludes that Spin achieves better performance than NuSMV for P system models. More details regarding the P system transformation into SMV, the language accepted by NuSMV, are given in [11, 13], which use the model checker counterexamples for test generation.

In [21] the probabilistic model checker Prism is employed to verify a stochastic P system. The paper presents a case study representing the cell cycle in eukaryotes, described using a P system specification which is translated into Prism. Specific questions are then formulated and run against the Prism specification of the P system. Daikon, a dynamic invariant detector, has been used
in a similar context to extract P systems properties, which were later validated by Prism [3].

The ProB model checker is employed in [10] to verify P systems translated into Event-B.

7 Conclusions and Future Work

This paper makes significant advances in the area of model checking based verification of P systems by considering P systems with active membranes, in particular P systems with cell division rules, having a bounded number of produced membranes. It devises a theoretical framework for model checking of P systems, by extending the previous work on this subject to P systems with active membranes. This is implemented in Spin for a complex case study involving a well-known NP-complete problem. For this example, a number of non-trivial properties of the model are formulated and verified. The case study is very relevant for our approach since division rules may be the source of drastic change in the membrane structure.

In future experiments we aim to address more complex properties of the example considered here and to identify invariants of various stages. More complex instances (for larger \( n \) and \( k \)) will be tested and improvements of the Promela implementation will be sought in order to deal with the state explosion problem. Future work will also involve automating the process of transforming the P system specification, given as a P-Lingua file, into a Promela model and applying this approach on other challenging, real life examples.

Acknowledgment The work of FI and RL was supported by CNCSIS - UE-FISCSU, project number PNII - IDEI 643/2008. The authors IPH and MPJ acknowledge the support of the project TIN200913192 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds, and the support of the Project of Excellence with Investigador de Reconocida Valía of the Junta de Andalucía, grant P08-TIC-04200. The authors would like to thank the anonymous reviewers for their comments and suggestions that allowed us to improve the paper.

References


Basic Concurrency Resolution in Clock-free P Systems

Sergiu Ivanov\textsuperscript{1,2}

\textsuperscript{1} Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chișinău MD-2028 Moldova
E-mail: sivanov@math.md

\textsuperscript{2} Technical University of Moldova, Faculty of Computers,
Informatics and Microelectronics,
Ștefan cel Mare 168, Chișinău MD-2004 Moldova

Abstract. Clock-free P systems are an almost completely asynchronous model of P systems in which each rule application lasts for a different time. This makes the processes in this model more similar to the chemical processes in the biological cell, but also poses similar design problems as in concurrent programming. In this paper an attempt is made to solve synchronisation problems in clock-free P systems using the approaches commonly used in concurrent programming.

1 Introduction

In the recent years high-performance parallel computing has become available to a stunning number of computer stations. Running several tasks at the same time is now possible on the majority of desktop PCs. Parallelism in itself, however, can only be used to solve very specific problems. Often synchronisation mechanisms are imperatively required between the independent threads of execution. In such situations critical resources, which must be used in cooperation, appear. This leads to concurrency problems. While most of such problems do not have stock solutions, they can frequently be classified or decomposed into a combination of classical concurrency problems.

The emergence of parallel desktop computing has transformed concurrency problems from situations confined to computing laboratories into problems which must be solved by any computer application developer who has the goal of creating a competitive product. Every programming framework which claims being complete provides synchronisation mechanisms which are nothing but ways of solving concurrency problems.

On the other hand, P systems have always been an inherently parallel computing device. Classic P system models have a global clock which makes synchronisation easier; however clock-free P systems exist. In such P systems the global clock is absent. This makes clock-free P systems very similar to modern-day highly parallel computing architectures. In this paper we will attempt to study how far the analogy goes and whether it is possible to reuse the achievements
in the domain of concurrent programming to easier design clock-free P systems solving a specific problem.

In its turn, solving concurrency problems in clock-free P systems may prove useful for concurrent programming. Rule applications in clock-free P systems are more similar to their biological counterparts and thus fit the model of a group of concurrent processes more naturally. The high-level abstraction which leads to non-deterministic time for each a rule to complete as well as for the P system to halt is, on the one hand, very specific. On the other hand, it really nicely reflects the situations in concurrent programming when the programmer has no way of knowing beforehand how long will certain sections of the program take to complete and, therefore, he can treat these times as random. Unsurprisingly, this is how such situations are usually treated in concurrent programming.

In this paper we will attempt to exploit this similarity and explore how well are the approaches usually applied in concurrent programming suited for making clock-free P systems design clearer and easier.

2 Concurrency: Clock-free P Systems vs. P Systems with a Global Clock

When talking about P systems, their intrinsic parallelism is often highlighted. Indeed, all P system models have compartments (or analogous constructions) which evolve in parallel. Moreover, the rules are usually applied in parallel as well. Thus, many P system models have at least two distinct levels of parallelism. However, as mentioned above, to solve real-world problems with such parallel architectures, we need to be able to solve concurrency problems first.

The majority of P system models have a convenient built-in component which facilitates synchronisation: the global clock. The models evolve in well-defined steps which occur strictly at the “ticks” of this global clock. This does not offer solutions to synchronisation problems for granted, but it certainly provides a useful reference frame for assuring the proper timing of parallel processes with respect to each other.

In this paper we will focus on the clock-free model which does not define a clock at all, and it does not seem possible to easily reduce this model to another one with a global clock. In this model each application of a rule takes a (randomly) different time, which is generally a real value. It is thus impossible to know when exactly an application of a rule will add its right-hand side into the region it is associated with. This makes rules more similar to the chemical processes in the cell, but also to the parallel tasks in a concurrent application.

It certainly requires considerably more effort to assure synchronisation between several processes in a clock-free P system than in a P system with a global clock. In this paper we will try to focus on creating some basic synchronisation mechanisms in clock-free P systems.
3 Preliminaries

3.1 P Systems

A clock-free membrane system is defined by a tuple

$$\Pi = (O, C, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0),$$

where

- $O$ is a finite set of objects,
- $C$ is a finite set of catalysts, $C \subseteq O$,
- $\mu$ is a hierarchical structure of $m$ membranes, bijectively labeled by $1, \ldots, m$; the interior of each membrane defines a region;
- the environment is referred to as region $0$,
- $w_i$ is the initial multiset in region $i$, $1 \leq i \leq m$,
- $R_i$ is the set of rules of region $i$, $1 \leq i \leq m$,
- $i_0$ is the output region.

The rules of a clock-free membrane system have the form $u \rightarrow v$, where $u \in O^+$, $v \in (O \times \text{Tar})^*$. In case of non-cooperative rules, $u \in O$. The target indications from $\text{Tar} = \{\text{here, out}\} \cup \{\text{in}_j \mid 1 \leq j \leq m\}$ are written in the following way: $(a, t)$, $a \in O$, $t \in \text{Tar}$ and the target here is typically omitted.

The rules are applied in a maximally parallel way: no further rule should be applicable to the idle objects. In case of non-cooperative systems all objects evolve by the associated rules in the corresponding regions (except objects $a$ in regions $i$ such that $R_i$ does not contain any rule $a \rightarrow u$, but these objects do not contribute to the result). Rules are non-deterministically chosen at each moment in time when a change occurs in the configuration of the P system. The process of choosing which rules should be applied does not take any time.

In this paper we will consider that rule applications work in the following way: at the start of application the multiset in the left-hand side of the rule is subtracted from the corresponding region. When a rule application is complete, the multiset in the right-hand side of the rule is added to the corresponding region. No other processes pertaining to the rule application occur during the application. The time between the start and the end of a rule application is random and may be different between different applications of the same rule.

Rules with catalysts represent a special kind of cooperative rules which may only have two objects in the left-hand side, one of which is re-introduced in the system by the application of the rule. For example:

$$ca \rightarrow c\gamma, c, a \in O, \gamma \in O^*$$

Here $c$ is the catalyst. Rules with multistable catalysts are rules in which the catalyst is allowed to change into a different catalyst:

$$ca \rightarrow c'\gamma, c, c', a \in O, \gamma \in O^*$$
In the case of bistable catalysts, $c$ may turn to $c'$ after having catalysed a reaction and $c'$ may turn into $c$, while transformations of $c$ and $c'$ to other symbols are not allowed in reactions where $c$ and $c'$ serve as catalysts.

In clock-free P systems with promoters/inhibitors we consider rules of the following forms:

- $u \rightarrow v|a$, $a \in O$ – this rule is only allowed to be applied when the membrane it is associated with contains at least an instance of $a$; $a$ is called the promoter of this rule;
- $u \rightarrow v|\neg a$, $a \in O$ – this rule is only allowed to be applied when the membrane it is associated with contains no instances of $a$; $a$ is called the inhibitor of this rule.

Rules do not consume the corresponding promoters/inhibitors. A rule may have both a promoter and an inhibitor at the same time, in which case it can only be applied when there is at least one instance of the promoter and no instances of the inhibitor in the region. Note also, that a single instance of an object may act as a promoter for more than one instance of rewriting rules during the same transition.

A sequence of transitions is called a computation. The computation halts when such a configuration is reached that no rules are applicable.

In this paper we will only allow rules with promoters and catalysts.

### 3.2 Concurrency

A resource is any object which can used (read or written) by an active entity in scope. A critical resource is an object which may happen to be accessed by more than one active entity at the same time.

A concurrency problem is a situation during the execution of a group of threads when more than one thread accesses a critical resource. A solution to a concurrency problem is a way to (partially) order the execution of the threads in the group in such a way that parallel accesses to the critical resource do not deteriorate the execution of the group of threads and do not distort the results of execution.

A barrier is a synchronisation method for a group of threads. A thread is not allowed to proceed from the point in the code where the barrier is declared until all other threads in the group have reached the barrier.

A semaphore is a synchronisation mechanism introduced by E. W. Dijkstra in [1]. It includes a counter initialised to the value $N$ which can be modified by one of the following operations:

- $P$ – checks if the current counter value is greater than 0; decrements it if it is greater than zero or waits until it gets greater than 0;
- $V$ – increments the counter if its value is less than $N$ and does nothing otherwise.

A lock is a semaphore with $N = 1$. 
4 Strategies

We will call a set transformation of $A$, $A \subseteq U$ any function $T_A : A \cup \{\emptyset\} \rightarrow 2^U$, where $U$ is the universal set of entities. A multiset transformation of $A$ is any function $M_A : A \rightarrow U^*$.

We define an application of the set transformation $T_A$ to a set $B \subseteq A$ in the following way:

$$T_A(B) = \begin{cases} \bigcup_{b \in B} T_A(b), & \text{for } B \neq \emptyset \\ T_A(\emptyset), & \text{for } B = \emptyset \end{cases}$$

The application of $M_A$ to a multiset $w \in A^*$ is defined in an analogous way, with the operation of set union being substituted with the operation of multiset union and the empty set being substituted with $\lambda$.

A non-conflicting set transformation of $A$ is such a transformation of $A$ that

$$T(a) \cap A \subseteq \{a\}$$

A non-conflicting multiset transformation is defined in an analogous way.

Informally, the application of a non-conflicting transformation to $B$ only adds new objects to $B$. This property is especially important for multisets, in which case application of a non-conflicting transformation is guaranteed to not modify the quantity of elements other than its argument.

We define a parametrised set transformation $T_A^P$ of a set $A$ to be any function $T_A^P : (A \cup \{\emptyset\}) \times P \rightarrow 2^U$, where $P$ is a set of valid parameter values. A parametrised multiset transformation $M_A^P$ is a function $M_A^P : (A \cup \{\emptyset\}) \times P \rightarrow U^*$.

A strategy for a P system $\Pi$ is defined by the following tuple:

$$S_\Pi = (T_O, M_{w_1}, M_{w_2}, \ldots, M_{w_m}, T_{R_1}, T_{R_2}, \ldots, T_{R_m}),$$

where $T_A$ are set transformations of the sets $A \in \{O\} \cup \{R_i \mid 1 \leq i \leq m\}$,

$M_{w_i}$ are non-conflicting multiset transformations of $O$ associated with the corresponding multisets $w_i, 1 \leq i \leq m$.

A parametrised strategy $S_\Pi^p, p \in P$ is a strategy in which all transformations are parametrised.

Strategies for catalytic P systems include a set transformation $T_C$ of the set of catalysts $C$.

An application of a strategy $S_\Pi$ to the corresponding P system $\Pi = (O, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0)$ is defined in the following way:

$$S_\Pi(\Pi) = (T_O(O), \mu, M_{w_1}(w_1), M_{w_2}(w_2), \ldots, M_{w_m}(w_m),$$

$$T_{R_1}(R_1), T_{R_2}(R_2), T_{R_m}(R_m), i_0)$$

An application of a parametrised strategy $S_\Pi^p$ to $\Pi$ is correspondingly defined as follows:

$$S_\Pi(\Pi, p) = (T_O(O, p), \mu, M_{w_1}(w_1, p), M_{w_2}(w_2, p), \ldots, M_{w_m}(w_m, p),$$

$$T_{R_1}(R_1, p), T_{R_2}(R_2, p), T_{R_m}(R_m, p), i_0)$$
Applications of strategies to catalytic P systems include the application of the corresponding set transformation $T_C$ of the set of catalysts to $C$.

A (parametrised) strategy can be perceived as a way to modify an aspect of the functionality of a P system. Strategies also offer the advantage of extensibility, because they allow building the necessary behaviour out of specific elements. In this paper we will consider synchronisation strategies which will assure the proper time ordering of the applications of rules a clock-free P system.

5 Synchronisation of Rule Applications

Due to the fact that in clock-free P systems different applications of the same rule may last differently, one of the basic problems is knowing when all applications of the rule $a \rightarrow \gamma$, $a \in O$, $\gamma \in O^*$ have finished and added their right-hand sides to the corresponding multisets. Note that the ability to detect this moment in time will re-establish a situation remotely similar to what happens in P system models with a global clock.

We will refer to the rule whose applications we would like to synchronise as the target rule.

In the following subsections we will consider the possible ways of solving the posed synchronisation problem. To simplify descriptions, we will only consider one-membrane transitional clock-free P systems in this section:

$$\Pi = (O, C, [\_], w, R, 1)$$

We will consider more complex membrane structures in subsequent sections.

5.1 Sequential Strategy

One of the most obvious ways to address any synchronisation problem is avoiding parallelism at all. This simplifies the problem greatly and often leads to concise and clear solutions. The principal drawback of this approach is that it literally wastes the parallel capabilities of the device it is implemented on.

For the sequential strategy we will demand that, for the target rule $a \rightarrow \gamma$ the following holds:

$$a \notin \gamma$$

A sequential strategy may be described in the following way:

$$\text{Seq}^R = (T_O^{R(S)}, T_C^{R(S)}, M_w^{R(S)}, T_R^{R(S)}),$$

where

$$T_O^{R(S)}(a, a \rightarrow \gamma) = \{a, f_1^a, f_2^a, f_3^a, p^a, s^a\},$$

$$T_O^{R(S)}(b, a \rightarrow \gamma) = \{b\}, b \in O \setminus \{a\},$$

$$T_O^{R(S)}(\emptyset, a \rightarrow \gamma) = \emptyset,$$

$$T_C^{R(S)}(\emptyset, a \rightarrow \gamma) = \{c^a\}.$$
\[ T^R_C(c_2, a \rightarrow \gamma) = \{ c_2, e^a \}, C = \{ c_2 \} \cup C_1, \]

\[ T^R_C(c_1, a \rightarrow \gamma) = \{ c_1 \}, c_1 \in C_1, \]

\[ M_w^R(a^n, a \rightarrow \gamma) = a^n p_a, \]

\[ M_w^R(b^m, a \rightarrow \gamma) = b^m, b \in W \setminus \{ a \}, \text{ where } W \text{ is the underlying set of } w, \]

\[ M_w^R(\lambda, a \rightarrow \gamma) = \lambda, \]

\[ T^R_R(a \rightarrow \gamma, a \rightarrow \gamma) = \{ e^a a \rightarrow e^a f_1^a f_2^a \gamma | p^a, p^a \rightarrow \lambda, f_1^a \rightarrow \lambda | a, f_2^a \rightarrow f_3^a, f_3^a \rightarrow p^a | a, f_3^a \rightarrow s^a | f_1^a, f_1^a \rightarrow \lambda | f_2^a \}, \]

\[ T^R_R(r, a \rightarrow \gamma) = \{ r \}, r \in R \setminus \{ a \rightarrow \gamma \}. \]

We will explicitly remark that \( T^R_C(C) = C \cup \{ e^a \}. \) The (S) notation indicates that the transformations pertain to the sequential strategy.

The rules and symbols added by \( \text{Seq}^R \) work as follows. The symbol \( e^a \) is the catalyst which assures that only one instance of the target rule \( a \rightarrow \gamma \) is applied at a time. The promoter \( p^a \) only allows this rule to be applied a single time, because once \( p^a \) appears in the system, it is immediately erased.

An application of the target rule brings about two verifier symbols \( f_1^a \) and \( f_2^a. \) \( f_2^a \) is immediately involved into the transformation into \( f_3^a. \) If there are more instances of \( a \) in the system, \( f_3^a \) is immediately employed by the rule \( f_1^a \rightarrow \lambda | a. \) It does not matter when this rule will complete, because the symbol \( f_3^a \) becomes unavailable exactly when this rule starts being applied and will never become available again. When the symbol \( f_3^a \) eventually appears in the system, two states are possible:

- there is no \( f_3^a \) (or it is unavailable and will never become available again)
- and there are still instances of \( a; \)
- there is an instance of \( f_3^a \) and no instances of \( a. \)

If there are still instances of \( a \) left, \( f_3^a \) produces a new instance of the promoter \( p^a \) to continue the applications of the target rule. If there is an instance of \( f_3^a, \) the rule \( a \rightarrow \gamma \) is not applicable any more. When \( f_3^a \) appears in the system, \( f_3^a \) starts being consumed by the rule \( f_1^a \rightarrow \lambda | f_2^a, \) so we can consider it is no longer present. However, at the same moment when \( f_3^a \) starts being erased, the rule \( f_3^a \rightarrow s^a \) is applicable and it is applied (because of the maximal parallelism). This will eventually lead to the appearance of \( s^a \) in the system, which should be perceived as the signal that all applications of the target rule have completed.

\( \text{Seq}^R \) does not obviously produce a meaningfully working system. One should add the rules to properly utilise the symbol \( s^a \) and to transition into the next phase of evolution of \( \Pi. \)

The sequential synchronisation strategy is based on the following simple principle: never allow more than one rule of the form \( \alpha \rightarrow \beta, \beta \neq \lambda \) to be applied at a time. Exception is made for the rules which only erase symbols because such rules make the symbols in \( \alpha \) unavailable immediately, and when such rules eventually complete, the symbols in \( \alpha \) do not become available again.
Later in the paper we will use only the symbol Seq to refer to the sequential synchronisation strategy.

We will now provide an example of how Seq can be applied.

Example 1 (Generate $a^{2n}$ with Seq). We will build a clock-free P system which non-deterministically generates $a^{2n}$ in the skin membrane [2].

In clocked P systems, the simplest way to generate $a^{2n}$ would be to employ rules $a \rightarrow a^2$. In clock-free P systems we cannot do that freely. Consider, for example, the situation when the multiset in the skin region is $a^2$. At this moment two applications of the rule $a \rightarrow a^2$ start. It may happen that one of the applications will finish very soon, the multiset in the skin will become $a^2$ (one instance of $a$ is busy in the second application of the rule). Another two applications of the rule $a \rightarrow a^2$ will start and, suppose, these two applications will complete at exactly the same time as the other long-lasting application. This will result in the skin containing $a^2 \cdot a^2 \cdot a^2 = a^6$, which is far from what we would like to have. The problem in this situation is, obviously, that the rules are not applied in well-defined steps and applications of the rule can consume both “newer” and “older” instances of $a$ in parallel.

We can address this problem by synchronising all applications of rules using Seq. We cannot apply Seq to a P system containing the rule $a \rightarrow a^2$, however, because $a \in a^2$. To solve this problem we will consider a P system with two rules:

$$\Pi_1 = \{(a, b), \emptyset, a, \{a \rightarrow b^2, b \rightarrow a^2\}, 1\}$$

We will have to synchronise both rules now. We will also have to build a connection between the two phases introduced by the two applications of Seq (strategies $S_{ab}^{(S)}$ and $S_{ba}^{(S)}$), as well add proper finalisation ($S_{fin}^{(S)}$):

$$\Pi_2 = S_{fin}^{(S)}(S_{ba}^{(S)}(\text{Seq}(S_{ab}^{(S)}(\text{Seq}(\Pi_1, a \rightarrow b^2)), b \rightarrow a^2)))$$

We will describe the three new strategies informally because they are very simple:

- $S_{ab}^{(S)}$ changes the symbol $s^a$ into $p^b$ and the rule $f_3^a \rightarrow s^a | f_3^a$ into $f_3^a \rightarrow p^b | f_3^a$;
- $S_{ba}^{(S)}$ changes the symbol $s^b$ into $p^a$ and the rule $f_3^b \rightarrow s^b | f_3^b$ into $f_3^b \rightarrow p^a | f_3^b$;
- $S_{fin}^{(S)}$ adds the symbol $s^h$, changes the multiset in the skin membrane into $ap^a$, and adds the rules

$$\{f_3^a \rightarrow s^a | f_3^a, f_3^b \rightarrow s^b | f_3^b, b \rightarrow a|_{s^h}, c^a \rightarrow \lambda|_{s^h}, c^b \rightarrow \lambda|_{s^h}\}$$

these lead to a non-deterministic halting and clean-up of $\Pi_2$.

The resulting P system thus looks as follows:

$$\Pi_2 = (O_2, C_2, \{1\}, w_2, R_2, 1), \text{ where}$$

$$O_2 = \{a, b, f_1^a, f_2^a, f_3^a, p^a, f_1^b, f_2^b, f_3^b, p^b, s^h\}$$

$$C_2 = \{c^a, c^b\}$$
\[ w_2 = ap^a \]
\[ R_2 = \{ c^a a \rightarrow c^a f_1^a f_2^a \gamma | p^a, p^a \rightarrow \lambda, f_1^a \rightarrow \lambda | a, f_2^a \rightarrow f_3^a, \\
 f_3^a \rightarrow p^a | a, f_3^a \rightarrow p^b | f_1^a, f_1^a \rightarrow \lambda | f_2^a \} \]
\[ \cup \{ c^b b \rightarrow c^b f_1^b f_2^b \gamma | p^b, p^b \rightarrow \lambda, f_1^b \rightarrow \lambda | b, f_2^b \rightarrow f_3^b, \\
 f_3^b \rightarrow p^b | b, f_3^b \rightarrow p^b | f_1^b, f_1^b \rightarrow \lambda | f_2^b \} \]
\[ \cup \{ f_3^a \rightarrow s^a | f_3^b, f_3^b \rightarrow s^b | f_1^a, b \rightarrow a | s^a, c^a \rightarrow \lambda | s^a, c^b \rightarrow \lambda | s^b \}. \]

This system has more rules than the system suggested in [2], but it is much easier to describe using a combination of strategies.

### 5.2 Barrier Synchronisation Strategy

In this subsection we will avoid serialising the applications of rules and will allow all the applications of a rule to run in parallel. We will need to know when exactly all applications have completed and only then allow the system to proceed to the subsequent phases of evolution. In concurrent programming such problems are resolved using the barrier synchronisation mechanism.

In computer programming a barrier is often implemented by counting the applications that have arrived at the checkpoint and blocking them until all of them have arrived. We will implement a similar approach in P systems:

\[ \text{Bar}^R = (T_O^{R(B)}, T_C^{R(B)}, M_w^{R(B)}, T_R^{R(B)}), \]
\[ a \rightarrow \gamma \in R, \]
\[ T_O^{R(B)}(a, a \rightarrow \gamma) = \{a, p^a, c_1^a, c_2^a, c_3^a, f_1^a, f_2^a, f_3^a, f_4^a, s^a\}, \]
\[ T_O^{R(B)}(b, a \rightarrow \gamma) = \{b\}, b \in O \backslash \{a\}, \]
\[ T_O^{R(B)}(\emptyset, a \rightarrow \gamma) = 0, \]
\[ T_C^{R(B)}(\emptyset, a \rightarrow \gamma) = \{c_1^a, c_2^a, c_3^a, f_1^a, f_3^a, s^a\}, \]
\[ T_C^{R(B)}(c, a \rightarrow \gamma) = \{c, c_1^a, c_2^a, c_3^a, f_1^a, f_3^a, s^a\}, C = \{c\} \cup C', \]
\[ T_C^{R(B)}(c', a \rightarrow \gamma) = \{c', c' \in C', \}
\[ M_w^{R(B)}(a^n, a \rightarrow \gamma) = a^n (c_1^a)^n p^a f_1^a f_3^a, \]
\[ M_w^{R(B)}(b, a \rightarrow \gamma) = b, b \in W \backslash \{a\}, \text{where } W \text{ is the underlying set of } w, \]
\[ M_w^{R(B)}(\lambda, a \rightarrow \gamma) = \lambda, \]
\[ T_R^{R(B)}(a \rightarrow \gamma, a \rightarrow \gamma) = \{a \rightarrow c_2^a \gamma | p^a, p_a \rightarrow \lambda, c_2^a c_1^a \rightarrow c_3^a, c_3^a \rightarrow \lambda, f_1^a \rightarrow f_3^a, \\
 f_2^a \rightarrow f_4^a | c_1, f_3^a f_4^a \rightarrow f_4^a f_2^a, f_3^a f_2^a \rightarrow s^a\}, \]
\[ T_R^{R(B)}(r, a \rightarrow \gamma) = \{r\}, (b \rightarrow \delta) \in R \backslash \{a \rightarrow \gamma\}. \]
We will use the symbol Bar to refer to this synchronisation strategy. The \((B)\) notation indicates that the transformations pertain to the Bar strategy. Note that in this strategy we employ bistable catalysts. Another remark is that \(a \in \gamma\) is allowed.

The rules and symbols introduced by this strategy work as follows. From the very beginning the number of instances of \(a\) (and thus, the number of rule applications to occur) is recorded in the quantity of the counter symbol \(c_1^a\). When one of the rule applications finalises, it produces an instance of \(c_2^a\). \(c_2^a\) always erases exactly one instance of \(c_1^a\) and transforms into \(c_3^a\), which is eventually erased as well.

The symbols \(f_1^a\) and \(f_2^a\) in the skin membrane are evolving at the same time. Their goal is to verify whether \(c_1^a\) is still present in the system. Whenever \(f_1^a\) is in the system, it immediately starts being transformed into \(f_2^a\). In parallel, \(f_2^a\) starts being transformed into \(f_3^a\) if there are instances of \(c_1^a\) in the system. When both \(f_3^a\) and \(f_4^a\) appear, the conclusion that \(c_1^a\) is still around is made and a new iteration of checking is started by reproducing \(f_1^a\) and \(f_2^a\). If, however, there is no \(c_1^a\) in the system, \(f_2^a\) remains idle and \(f_3^a\) finds \(f_2^a\) instead of \(f_4^a\). This is correctly interpreted as the sign that there are no more instances of \(c_1^a\) in the membrane and the signal symbol \(s^a\) is produced.

Note that the main idea behind this strategy is to keep as little as possible functionality associated with the parallel execution of the target rule and avoid as much interaction between the rules as possible. A completion of each rule application only causes a removal of an instance of \(c_1^a\) and cannot interfere with other rule applications. The verification of the barrier condition is done in a “master thread” which runs sequentially, independently of the rule applications. This gives us much more control over the situation. Nevertheless, the applications of the target rule are carried out in parallel.

The following theorem states that, using the counting approach seen in Bar, it is impossible to achieve barrier synchronisation with simple catalysts:

**Theorem 1 (Necessity of bistable catalysts).** It is impossible to erase the counter symbols \(c_1^a\) using only rules with catalysts and promoters.

**Proof.** Bar compares the quantities of \(c_1^a\) and \(c_2^a\) which represent, respectively, the number of already completed and yet incomplete rule applications. The comparison is done by erasing a pair \(c_1^a, c_2^a\) immediately as a rule has completed its application. If we do not use bistable catalysts and use simple catalysts \(c_1^a\) and \(c_2^a\) (possibly \(c_2^a = c_5^a\)) to erase \(c_1^a\) and \(c_2^a\) correspondingly, we will need to delete the catalysts after they have done their jobs. We cannot erase these catalysts using non-cooperative rules with promoters only because it may happen that a parallel rule application will add an erasing promoter to the system before \(c_1^a\) and \(c_2^a\) will have done their job; this may lead to premature erasure of one of the catalysts and the failure of the system to do proper comparison. If we erase the catalysts using catalytic rules (with or without promoters), we will need to erase the new pair catalysts, which is exactly the same problem.

On the other hand, erasing \(c_1^a\) with non-cooperative rules will obviously immediately erase all instances of \(c_1^a\).
Note that it does not matter how one erases $c^2_1$; the principal problem is that an attempt to erase $c^2_1$ with non-cooperative rules will erase all instances of this symbol, while attempting to erase it with simple catalytic rules only changes the problem into the one of erasing the catalyst.

**Example 2 (Generate $a^{2n}$ with Bar).** We will build another clock-free P system which non-deterministically generates $a^{2n}$ in the skin membrane [2].

We have already solved this problem using a combination of Seq strategies. In this example we will show how to solve it using the Bar strategy. In the case of Bar, we can use the rule $a \rightarrow a^2$, but the problem is that, before the applications start, we need to have as many $c^1_1$ in the system as there are instances of $a$. This is easily solved at initialisation, but will have to be explicitly maintained during the evolution of the P system. The obvious way to assure the proper presence of the counter symbol would be producing an instance of the counter symbol per one instance of $a$ produced. This poses the problem of distinguishing between the counter symbols belonging to different generations. The easiest way to solve this problem is, again, splitting the evolution of the system into a sequence of two phases, with different counter symbols in each phase. This would mean applying the Bar twice and do some further normalisation, just as in Example 1.

Before doing this, however, we will observe that the rules $c^1_1 \rightarrow \lambda$ and $f^a_1 \rightarrow f^a_2$ do not depend on the change of the counter symbol. These rules could be reused by any number of Bar strategies provided that no two of them ever run in parallel with respect to each other. We will further refer to the barrier strategy which reuses the two common rules as Bar$'$.

We will start with the following clock-free P system:

$$H_3 = \{(a, c^a_1, c^b_1), \emptyset, c^a_1, \{a \rightarrow (c^a_1)^2 a^2, a \rightarrow (c^b_1)^2 a^2\}, 1)$$

Note that we have used the letter $b$ only to keep the analogy with the corresponding constructions from Example 1.

The P system properly generating $a^{2n}$ using Bar and Bar$'$ is constructed as follows:

$$H_4 = S^{(B)}_{\text{fin}}(S^{(B)}_{\text{ba}}(\text{Bar}(S^{(B)}_{\text{ab}}(\text{Bar}(H_2, a \rightarrow (c^a_1)^2 a^2), a \rightarrow (c^b_1)^2 a^2))))$$

Again, we will only describe the three auxiliary strategies informally:

- $S^{(B)}_{\text{ab}}$ changes the symbol $s^a$ into $p^b$ and changes the rule $f^a_3 f^a_2 \rightarrow s^a$ into $f^b_3 f^b_2 \rightarrow f^b_1 p^b f^b_2$;
- $S^{(B)}_{\text{ba}}$ changes the symbol $s^b$ into $p^a$ and changes the rule $f^a_3 f^a_2 \rightarrow s^a$ into $f^b_3 f^b_2 \rightarrow f^b_1 p^a f^b_2$ (remember that the symbols $f^a_3$ and $f^a_2$ are reused by Bar$'$);
- $S^{(B)}_{\text{fin}}$ changes the initial multiset into $ac^a_1 p^b f^a_1 f^a_2$ and adds the symbol $s$ and the rules $\{f^a_3 f^a_2 \rightarrow s, f^a_3 f^b_2 \rightarrow s, s \rightarrow \lambda\}$, which non-deterministically stop the execution of the system after another phase has completed.
The resulting clock-free P system $\Pi_4$ looks as follows:

$$\Pi_4 = (O_4, C_4, [\ ]_1, w_4, R_4, 1),$$

where

\[ O_4 = \{a, p^a, c_1^a, c_2^a, c_3^a, f_1^a, f_2^a, f_3^a, f_4^a, p, c_1^b, c_2^b, f_1^b, f_2^b\}, \]

\[ C_4 = \{c_2^a, c_3^a, c_2^b, f_1^a, f_3^a, s\}, \]

\[ w_4 = ac_1^a p^a f_1^a f_2^a, \]

\[ R_4 = \{a \rightarrow c_2^a (c_1^b)^2 a^2 \mid p, p^a \rightarrow \lambda, c_2^a c_1^a \rightarrow c_3^a, c_3^a \rightarrow \lambda, f_1^a \rightarrow f_3^a, f_2^a \rightarrow f_4^a | c_1^a, \]

\[ f_3^a f_2^a \rightarrow f_1^a f_2^a, f_3^a f_2^a \rightarrow p^b f_1^a f_2^b, \]

\[ \cup \{a \rightarrow c_2^a (c_1^b)^2 a^2 | p, p^b \rightarrow \lambda, c_2^b c_1^b \rightarrow c_3^b, f_2^b \rightarrow f_3^b | c_1^b, f_3^b f_1^b \rightarrow f_4^b f_2^b, \]

\[ f_3^b f_2^b \rightarrow p^a f_1^b f_2^b \} \cup \{f_3^a f_2^a \rightarrow s, f_3^a f_2^a \rightarrow s, s \rightarrow \lambda\}. \]

6 Synchronisation of Processes

While rules are the obvious targets for synchronisation, they are badly fit for modelling actual processes. The only similarity we could outline was the fact that we cannot know exactly how much time a certain rule application will take. There is a series of traits which make rule applications different from a general notion of process, including:

- atomicity – the only moment when concurrency appears is the rule selection phase, which does not conceptually belong to rule application; rule applications themselves do not interact and the effects of concurrency are only observable between rule applications; once a rule application has started, nothing can alter its course or effect;
- logical independence – rules are not otherwise logically connected than by the fact that they may employ the same symbols; keeping the focus on rule applications as models of parallel processes drives us away from considering more complex dependence which may occur between real processes.

Note that these differences are also relevant while comparing rule applications to the chemical processes in the biological cell.

In this section we will try to go beyond this limitation by considering P systems of the following general form:

$$\Pi_{par} = (O, C, [\ ]_1, [\ ]_2 \ldots [\ ]_m, R, w_s, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, s)$$

This P system has $m$ membranes, numbered 1 through $m$, contained within the skin membrane with label $s$. Each of these membranes will model a parallel process and we will develop the fundamental synchronisation mechanisms, allowing the resolution of concurrency problems which may appear between the processes.

Note that, although $\Pi_{par}$ has only one level of nested membranes, it may be used to address synchronisation of more complex membrane structures by picking a certain membrane, considering its immediately inner membranes, and ignoring
the inner structure of the latter membranes. This approach allows treating concurrency problems consistently by conceptually isolating independent activities. Observe, though, that we do not attempt to optimise the solutions as yet.

We will call the symbols located in the inner membranes of $\Pi_{\text{par}}$ local data, while the symbols contained with the skin will be referred to as shared data. An inner membrane can modify the shared data by emitting specific symbols. We do not restrict the modifications these specific symbols may carry out, but instead require that they spend as little time as possible in the skin membrane. In other words, we expect the bulk of computations to happen within the inner membranes.

6.1 Lock

The lock is one of the most important synchronisation solutions used in parallel and concurrent programming. Locks are used to protect data from corruption and are a rather simple concept to grasp. The situation in P systems is not that straightforward though.

The notion of corruption is irrelevant for local data. Defining corruption for shared data in $\Pi_{\text{par}}$ is not trivial, because, as noted above, rule applications are already inherently atomic. However, if there is information encoded in a multiset of symbols (not one symbol) and there is more than one rule that can modify it, the situation when two or more rules modify parts of this information at the same step may be undesirable. As a very simple example consider the situation when a membrane contains the multiset $a^n$ and has the rules \{$a \rightarrow a^2$, $a \rightarrow a^3$\} associated. If, at the next step, we need the region to contain either $a^{2n}$ or $a^{3n}$, we only want one of the two rules to be applied, not both of them. This situation is easily solved at the level of rules, but is not that simple when entire membranes are involved. We will use locks to handle such scenarios.

To implement a lock, we have two immediately visible options: either modify the state of the data itself or have a separate state. Since the data is often encoded in the multiplicity of certain symbols, modifying its state requires more than one rule application, which poses another synchronisation problem. Therefore, we will follow the second route of having a separate state — we will introduce a special lock symbol $\$$ in the skin membrane.

The sequence of actions for process $i$ when it wants to modify shared data is the following:

1. if data is locked, wait until it is unlocked;
2. lock the data;
3. modify the data;
4. unlock the data.

The process needs to read the state of the lock. In the model of clock-free P systems we consider in this paper, this can be done either by having the lock symbol $\$$ promote a rule application or employing it in a catalytic rule. The first approach is inapplicable, because a promoter may promote more than one rule
at the same time and therefore two processes may find the data unlocked at the same step, which is precisely what we need to avoid.

To implement the second approach, we will have each membrane $i$ emit a specific reader symbol $r_i$. We will also add to the skin membrane the rules

\[
\{ r_i \$_i \rightarrow r'_i \$_i, r'_i \rightarrow (s_i, i), \$_i \rightarrow \lambda \mid 1 \leq i \leq m \}
\]

No matter how many membranes have emitted the reader symbol, only one $r_i$ will get a chance to lock the data by changing $\$_i$ into $\$_i$. $r'_i$ will later send the symbol $r_i$ into the corresponding membrane in the form of $s_i$, which signals that membrane $i$ has access to the shared data. When the membrane $i$ has finished processing the data, it will emit the symbol $\$_i$ into the skin, thus unlocking the data. We will explicitly remark that neither of the enumerated rules will be applied more than once at a time and thus clock-freeness has no effect over manipulations on the lock.

Note that we could leave the symbols $\$_i'$ in the skin membrane, because they have no effect at all.

6.2 Semaphore

The semaphore is another important synchronisation primitive, used to protect data and to assure proper ordering of processes. Semaphores and locks are strongly related and, in our implementation, we will use the same idea as for implementing the lock.

In order to add a semaphore to the skin region we will add $\$_N$ to its content. To lock the semaphore (carry out $P$), the membrane $i$, $1 \leq i \leq m$, will eject the symbol $p_i$ into the skin. The corresponding rules which should be added to the skin are:

\[
\{ p_i \$_i \rightarrow p'_i \$_i, p'_i \rightarrow (s_i, i), \$_i \rightarrow \lambda \mid 1 \leq i \leq m \}
\]

Note that these are exactly the same rules as the ones used to manipulate the locks, which stresses the similarity between locks and semaphores.

As expected, to carry out the $V$ operation and unlock the semaphore, the membrane $i$ should emit an instance of $\$_i$ into the skin.

The only difference between the implementation of the lock and the semaphore is in the number of instances of $\$_i$ in the skin region. This corresponds to the conception in concurrent programming that locks are semaphores with $N = 1$ and it is remarkable how this relationship is directly reflected in this implementation of semaphores and locks in P systems.

7 The Dining Philosophers Problem

The Dining Philosophers problem is a classic concurrency problem. One of the most well-known setups for this problem is shown in Figure 1. The situation is as follows [3]: there are five philosophers who sit at the table in front of a plate of spaghetti. At a certain moment of time, a philosopher either eats or thinks.
Each philosopher needs two chopsticks to eat the spaghetti on his plate, but there are only five chopsticks, so they cannot eat all five at the same time. The philosophers do not take the two forks at the same time, but rather one after the other. When a philosopher grabs a chopstick, he does not release it until he has eaten. If a philosopher wants to eat but doesn’t have enough chopsticks, he waits. The problem is to assure such a fork taking strategy that all the philosophers have a chance to eat and more than one philosopher may eat at a time.

One of the classical solutions is to allocate a lock per fork and to not allow more than four philosophers to start taking forks at a time. To solve this problem we will adopt the Π_{par} model: we will map a philosopher to an inner membrane and will therefore have five inner membranes. Each fork will be implemented as a lock and there will be a semaphore the philosopher membranes will have to pass in order to start taking forks.

Initially, all philosophers are thinking, and the multiset in the inner membrane \( i \), \( 1 \leq i \leq 5 \), is \( t \). The set of rules associated with the membrane \( i \) is:

\[
R_i = \{ t \rightarrow (p_i, out), e_1 \rightarrow (r_{il}, out), e_2 \rightarrow (r_{ir}, out), e \rightarrow t(c, out)(f_i, out)(f_{i+1}, out) \}
\]

The skin membrane initially contains the symbols \{f_i | 1 \leq i \leq 5\} \cup \{c^4\}. The symbols \( f_i \) are the locks implementing the forks, while the symbols \( c \) are the entering semaphore. The rules associated with the skin membrane are the following:

\[
R_s = \{ p_i c \rightarrow p'_i c, p'_i \rightarrow (e_1, i), c_i \rightarrow \lambda | 1 \leq i \leq m \} \\
\cup \{ r_{il} f_i \rightarrow r'_{il} f', r'_{il} \rightarrow (e_2, i), f'_i \rightarrow \lambda | 1 \leq i \leq m \} \\
\cup \{ r_{ir} f_{i+1} \rightarrow r'_{ir} f'_i f_{i+1}, r'_{ir} \rightarrow (e, i), f'_{i+1} \rightarrow \lambda | 1 \leq i \leq m \}
\]

A philosopher membrane thinks for an indefinite amount of time (the application of the first rule in \( R_i \)) and then ejects a \( p_i \) into the skin, to try to lock
the semaphore. If the semaphore can still be locked (there are less than four
philosophers eating at the moment), \( p_1 \) locks the semaphore and sends \( c_1 \) back
into the membrane \( i \). The membrane then sends an \( r_{il} \) into the skin, to take
the fork on the left. If this fork is available, it is taken and \( c_2 \) is sent back into the
membrane \( i \). The membrane then sends \( r_{ir} \) into the skin, to take the fork on the
right. When this fork has been successfully taken, the philosopher membrane
receives the symbol \( e \) which signals the transition into the eating state. The eating
state lasts for a random amount of time (the duration of the application of the
last rule in \( R_i \)) and then the philosopher membrane transitions into the thinking
state by changing \( e \) into \( t \) and sending \( c, f_i \), and \( f_{i+1} \) out into the skin to unlock
the semaphore and release the forks.

The set of catalysts employed in this system is the following:

\[
C = \{ p_i, p'_i, r_{il}, r'_{il}, r_{ir}, r'_{ir} \}
\]

Note that, according to the requirement stated at introduction of \( \Pi_{par} \), only
locking and unlocking of semaphores and locks is done in the skin region. Eating
and thinking is decided and performed within the corresponding philosopher
membranes.

8 Conclusion

This paper is an attempt to approach the processes in clock-free P systems from
the standpoint of parallel and concurrent programming. We applied ideas similar
to the ones used in programming to parallel processes in clock-free P systems.
The results of this attempt are interesting and hopefully useful, however, a con-
siderable number of open questions and research directions is still left.

In this paper only two very basic solutions to synchronisation problems have
been provided. These solutions could be used to solve instances of classic concurrency
problems whenever they appear in the process of designing a certain
clock-free P system. It is however unclear how to strictly classify concurrency
problems appearing during the design of a clock-free P system, because the clas-
sical problems are formulated for a different paradigm.

The other question is which of the classical concurrency problems make sense
in the context of clock-free P systems. The time a rule application lasts in such
a P system is an abstract value. This makes the criterion of comparing the speed
of execution of different solutions very difficult to define. It may thus only make
sense to use employ the sequential approach.

On the other hand, clock-free P systems may be used as a way to model other
processes, including parallel threads of execution in concurrent programming. In
this case striving to maintain as much parallelism as possible is a perfectly valid
goal.

A very important question is the distinction between clock-free and time-
free P systems [4]. The two models are not at all unrelated and, therefore, the
results shown in this paper may be applicable to time-free P systems as well. It
is necessary however to formally verify the possibility in the first place.
It would also be relevant to examine similar synchronisation questions in asynchronous P systems, in which the moment a rule application starts is undefined, as different from clock-free P systems.

Another important problem is the features used in the suggested implementations. We found bistable catalysts to be effective in constructing solutions to synchronisation strategies; the question of whether it is possible to do with simple catalysts remains open.

References

On the Efficiency of Tile Pasting $P$ System
with Active Membranes

S. Jebasingh$^1$, T. Robinson$^2$, and Atulya K. Nagar$^3$

1 Department of Mathematics, Saveetha Engineering College, Chennai, India 
   jebasingh21@yahoo.com
2 Department of Mathematics, Madras Christian College, Chennai, India 
   robin.mcc@gmail.com
3 Department of Computer Science, Liverpool Hope University, United Kingdom 
   nagara@hope.ac.uk

Abstract. Pasting System (PS) and its variants Extended Pasting Scheme (EPS) and Tabled Pasting System (TPS) are syntactic methods to generate tiling patterns using pasting rules. Tile Pasting $P$ System (TPPS) and its variant Tile Pasting $P$ System with active membranes (TPPS-AM) are new computational models, based on the structure and functioning of living cells, to generate tiling patterns and tessellation using simple rules of gluing tiles at their edges. In this paper we discuss the efficiency of TPPS-AM in generating the tiling patterns with the basic models.

Keywords. Syntactic methods, two-dimensional pattern, Pasting System, Picture Languages, $P$ system.

1 Introduction

The Pasting System (PS) and its variants such as Extended Pasting Scheme (EPS) and Tabled Pasting System (TPS) are syntactic methods, introduced by Robinson et al. in [9, 12], to generate tessellation and tiling patterns. These techniques use the notion of a pasting rule [9] that allows the edges of the corresponding tiles to get glued or attached at the specified edges and thus generates interesting tiling patterns. With reference to the generative capacity of these systems, it has been proved in [12] that pasting system and extended pasting scheme are incomparable and pasting system is properly contained in tabled pasting system.

Membrane computing (or $P$ System) [1], a branch of nature inspired systems is an area of computer science aiming to abstract computing models from the structure and functioning of living cells. A $P$ System consists of a membrane structure, multisets of objects placed inside the membranes, evolution rules governing the modification of these objects in time, and transfer of objects from one membrane to another membrane (inter-cellular communication). In recent years, generation of two dimensional picture languages using $P$ Systems is extensively studied. $P$ Systems generating arrays and Indian folk designs (kolam patterns) are considered in [16]. Tissue like $P$ Systems with active membranes
that generate local picture languages are considered in [2]. Also self assembly P
systems that self assemble together a graph structure using bond making rules
are studied in [3, 8, 13].

Recently, a new computational model, namely, Tile Pasting P System and its
variant Tile Pasting P System with active membranes, based on the structure
and functioning of living cells, is introduced in [12, 14] for generating two dimen-
sional tiling and tessellation patterns, that are formed by gluing regular polygon
tiles. In this paper, after discussing the notions related to pasting systems [14] in
section 2, we recollect, in section 3 the definition of TPPS and TPPS-AM which
was introduced in [12]. In section 4 we discuss the efficiency of TPPS-AM with
different combinations of developmental rules with the earlier models PS, EPS,
TPS and TPPS.

2 Preliminaries

A tile is a 2-dimensional topological region (disk) with distinguished boundary
points as vertices determining the boundary edges. A tiling in the Euclidean
plane is a countable family of tiles that cover the plane without gaps or overlaps.
Hence the intersection of any finite set of tiles in a plane tiling has zero area. Such
an intersection will consist of a set of points (vertices) and lines (edges). A tiling
is said to be non-periodic if it admits no translation. Two adjacent tiles have an
edge in common. A tile, its vertices and edges may be labeled distinctively.

A pasting rule [9, 12] is a pair \((x, y)\) of edge labels of two (not necessarily
different) tiles \(a\) and \(b\). If \(x\) is the label of the right (respectively left) edge of \(a\)
and \(y\) is the label of the left (respectively right) edge of \(b\), then an application
of the rule \((x, y)\) pastes side by side (or joins edge to edge) the two tiles \(a\) and \(b\).
We can likewise define pasting of two tiles one above the other. Tiling patterns
are thus made up of tiles, ‘glued’ (or pasted edge to edge) together.

**Definition 1.** A Pasting System (PS) [9, 12] is \(S = (\Sigma, P, t_0)\), where \(\Sigma\)
is a finite non empty set of (labeled regular polygons) tiles, \(P\) is a finite set of pasting
rules and \(t_0\) is the axiom pattern of tiles.

The set of all patterns generated from the axiom tile \(t_0\) constitutes the pattern
language \(L(G)\) of PS.

A pattern \(p_2\) is generated from a pattern \(p_1\) by applying the pasting rules in
parallel to all the boundary edges of \(p_1\) where pasting is possible. Note that the
labels of pasted edges in a pattern are ignored once the tiles are pasted. Rotation
of tiles is not allowed in the PS while pasting two tiles.

**Example 1.** Consider the Pasting System \(G_1 = (\Sigma, P, t_0)\) where

\[
\Sigma = \{ R, G, B, Y \}
\]

is the set of eight Wang tiles with edge labels \(R, B, G\) and \(Y\). The pasting rules
are given by \(P = \{(R, R), (G, G), (B, B), (Y, Y)\}\). For an axiom tile \(t_0\) with label
1 in \(\Sigma\), the third member of \(L(G_1)\) is shown in Fig. 1.
A variation of pasting system has been introduced in [9] by defining Extended Pasting Scheme (EPS) wherein the application of pasting rules is done either sequentially or in parallel and the sequence of edge labels of the boundary of a pattern generated is required to satisfy a constraint.

**Definition 2.** An Extended Pasting Scheme (EPS) is $G = (\Sigma, P, t_0, \Delta)$, where $\Sigma$ is a finite non empty set of (labeled regular polygons) tiles, $P$ is a finite set of pasting rules, $t_0$ is the axiom tile and $\Delta$ is a finite set of constraints on the edge labels of $\Sigma$.

Here $\Delta$ is a regular expression over the labels of the regular polygons/tiles. Patterns satisfying the regular expression are accepted in the language. The set of all patterns, generated from the axiom tile $t_0$, satisfying the constraint $\Delta$, constitutes the pattern language $L(G)$ of EPS.

**Example 2.** Consider the extended pasting scheme $G_2 = (\Sigma, P, t_0, \Delta)$ where

$\Sigma = \{\star, \bigtriangleup, \triangle, \bigcirc\}$

$P = \{(R,B)\}$, axiom tile $t_0 = \bigtriangleup$

$\Delta = \{B^+\}$

The language of tiling pattern generated in parallel is given as below.

$L(G_2) = \{\star, \bigtriangleup, \bigcirc, \ldots\}$. 

**Definition 3.** A Tabled Pasting System (TPS) [12] is $TS = (\Sigma, P, t_0, C)$, where $\Sigma$ is a finite non empty set of (labeled regular polygons) tiles, $P$ is a finite set of
tables \{t_1, t_2, \ldots, t_k\} and each table \(t_i\) contains pasting rules to glue the tiles, \(t_0\) is the axiom pattern and \(C\) is a control over \(P\).

The table of rules are successively applied to the tiles/tiling patterns obtained from the axiom tile. The tiles can be adjoined to the tiles/tiling patterns added by previous rules in the table. The set of all patterns generated from the axiom tile \(t_0\) by the successive application of tables of pasting rules according to the control \(C\) over the tables, constitutes the pattern language \(L(G)\) of TPS.

Example 3. Consider the tabled pasting system \(TS = (\Sigma, P, t_0, C)\) where

\[
\begin{align*}
\Sigma &= \{R_2, R_6, G_2, G_6, G_3, G_5, B_1, B_5, B_4, B_6\} \\
P &= \{t_1, t_2, t_3\} \\
t_1 &= \{(R_2, G_5), (R_4, G_1), (R_6, G_3), (G_6, B_3), (G_4, B_1), (G_2, B_5)\} \\
t_2 &= \{(B_1, G_4), (B_2, R_5), (B_3, G_6), (B_5, R_1), (B_5, G_2), (B_6, R_3)\} \\
t_3 &= \{(R_1, B_1), (R_2, G_5), (R_3, B_6), (R_4, G_1), (R_5, B_2), (R_6, G_3)\} \\
t_0 &= \{R_2, R_6, R_3, R_4\} \\
C &= \{(t_1t_2t_3)^n/n \geq 1\}
\end{align*}
\]

The tessellation obtained is shown in the Fig. 2.

![Fig. 2. A tessellation pattern generated by TPS](image)

### 3 Tile Pasting P System (TPPS)

**Definition 4.** A Tile Pasting P System (TPPS) [14] is a construct

\[
\Pi = (\Sigma, \mu, F_1, \ldots, F_m, R_1, \ldots, R_m, i_0)
\]

where \(\Sigma\) is a finite set of tiles; \(\mu\) is a membrane structure with \(m\) membranes labeled in a one-to-one way with \(1, \ldots, m\); \(F_1, \ldots, F_m\) are finite sets of picture
patterns over tiles of $\Sigma$ associated with the $m$ regions of $\mu$; $R_1, \ldots, R_m$ are finite sets of pasting rules defined in the form $\{(x, y) \in X/X \in i_{i_0}, \text{here, out}\}$, associated with the $m$ regions of $\mu$ and $i_{i_0}$ is the output membrane.

A computation in a TPPS is defined in a way similar to the array-rewriting P system [14] with the successful computations being the halting ones; to each picture pattern, from each region of the system, for which a pasting rule could be applied, should be applied. The picture pattern obtained by the pasting step is placed in the region indicated by the target associated with the rule used (here means that the pattern remains in the same region, out means that the pattern exits the current membrane and in means that the pattern is sent to one of the lower membranes $m_j$, if no internal membrane exists, then a rule with the target indication in cannot be used). If a set $R_i$ contains pasting rules with two different target indication, then the set of pasting rules associated with one target is chosen non-deterministically for application. A computation is successful only if it stops; a configuration is reached where no pasting rule can be applied to the existing picture patterns. The result of a halting computation consists of the picture patterns composed only of patterns placed in the membrane with label $i_{i_0}$ in the halting configuration. The set of all such picture patterns computed or generated by a TPPS $\Pi$ is denoted by $PL(\Pi)$. The family of all such languages $PL(\Pi)$ generated by systems as above, with at most $m$ membranes, is denoted by $PL_m(\Pi)$.

**Definition 5.** A tile pasting $P$ system with active membranes (TPPS – AM) [12] is a construct of the form

$$\Pi = (O, H, \mu, (t_{i_0}), \Sigma, T_k, (k \in \{1, \ldots, n\}), R_i, m_j)$$

where

- $O$ is an alphabet of labels of the edges of the tiles
- $H$ is a finite set of labels for the membranes
- $\mu$ is the membrane structure, consisting of $m$ membranes labeled from the set $H$ (not necessarily in a one-one manner)
- $(t_{i_0})$, $i \in \{1, \ldots, m\}$ is the axiom tile present in the membrane $m_i$.
- $\Sigma$ is a finite set of tiles associated with the regions of the membranes $m_i$, $i \in \{1, \ldots, m\}$.
- $T_k$, $k \in \{1, \ldots, n\}$, are the finite sets of tables containing evolution rules. An evolution rule is a pair $(\alpha, \beta)$, $\alpha, \beta \in O$, concerned with a pair of edges of tiles, which allows the edges of the corresponding tiles to get glued (pasting rule).
- $R_i$ is a finite set of developmental rules associated with the membrane $m_i$. The developmental rules can be in the following form:

  (a) Object evolution rule: $[p_i] \rightarrow T_k [p_{i+1}]$

  The evolution rules in $T_k$ are applied to the edges of the pattern $p_i$, present in the membrane $m_i$, and the resultant pattern $p_{i+1}$ is retained in the same membrane. Here the membranes are neither taking part in the application of these rules nor are they modified by them.
Object evolution and communication/creation rule: 

\[
[ i p_i ]_i \rightarrow T_k [ j p_{i+1} ]_j
\]

The effect of the rule is, the evolution rules contained in table \( T_k \) are applied to the pattern \( p_i \) present in the membrane \( m_i \) and the resultant pattern \( p_{i+1} \) is transferred to the membrane \( m_j \) (symport communication). The rule is applied only if both \( m_i \) and \( m_j \) are elementary membranes. If membrane \( m_j \) does not exist, then it is created adjacent to \( m_i \) (membrane creation) and the pattern is transferred to it.

Evolution rule with dissolution:

\[
[ i p_i ]_i \rightarrow T_k \delta p_{i+1}
\]

If a rule of the form \( T_k \delta \) is used in a region \( m_i \), then the evolution rules in table \( T_k \) are applied to the edges of the pattern \( p_i \), present in the membrane \( m_i \), and the resultant pattern \( p_{i+1} \) is placed in the region, directly outside the membrane, and the membrane \( m_i \) is dissolved as a consequence of \( \delta \), where \( \delta \notin O \) is a special symbol (dissolving capability). The skin membrane cannot be dissolved.

Out communication rule:

\[
[ i p_i ]_i \rightarrow T_k [ i ]_i p_{i+1}
\]

The evolution rules in \( T_k \) are applied to the edges of the pattern \( p_i \), present in the membrane \( m_i \), and the resultant pattern \( p_{i+1} \) is sent out to the region, directly outside the membrane \( m_i \). If this rule is applied to the skin membrane then the pattern sent out of the skin membrane is lost from the system.

\( m_j \), is the output membrane.

The computation starts with the membrane \( m_i \), containing the axiom pattern \( t_0 \). The evolution rules present in the tables \( T_k \) are applied to the edges of the tiles in a maximally parallel manner. For instance during a computation if pasting is possible at all the edges of the given pattern \( p_i \) then the evolution rules present in the table \( T_k \) are applied to all possible edges simultaneously. If \( R_j \) contains two or more developmental rules, then any one of them, chosen non deterministically is applied in the membrane. A sequence of transitions forms a computation and the result of a halting computation is the set of patterns sent to the output membrane.

The set of all such picture patterns computed by a TPPS - AM is denoted by \( \Pi(TPPS-AM) \). The family of all such languages \( \Pi(TPPS-AM) \) generated by systems defined as above, with at most \( m \) membranes and developmental rules of the form \( \alpha, \beta \in \{ \text{object evolution rule, object evolution and dissolution, object evolution and communication/creation, out communication rule} \} \), is denoted by \( \Pi_m(TPPS-AM, \alpha, \beta) \).

4 Results

In the following, we compare the generative power of PS, EPS, TPS, and TPPS with TPPS-AM. Let \( F_{PS}, F_{EPS}, F_{TPS} \) and \( PL_n(H) \) denote the families of languages/tiling patterns generated by Pasting System (PS), Extended Pasting Scheme (EPS), Tabled Pasting System (TPS) and Tile Pasting P System (TPPS) respectively.

Theorem 1. \( F_{PS} = \Pi_2(TPPS-AM, \alpha, \beta) \) where \( \alpha \) is the object evolution rule and \( \beta \) is the object evolution and dissolution rule.
Proof. Consider a Pasting System \( G = (\Sigma, P, t_0) \). A Tile Pasting \( P \) System with active membranes, having developmental rules in the form of object evolution and object evolution & dissolution rule, to generate the language \( L(G) \) is defined below.

\[ \Pi = (O, H, \mu, (t_0)_2', \Sigma', T_k, R_i, m_1) \]

where \( O \) is an alphabet of labels of the edges of the tiles of \( \Sigma \), \( H = \{1, 2\}, \mu = [1|2|2], (t_0)_2' = t_0, m = 2 \) is the initial degree of the system, \( \Sigma' = \Sigma, T_1 = P, R_1 = \{\phi\}, R_2 = \{[2p_1]_2 \rightarrow_{T_1} [2p_2]_2, [2p_1]_2 \rightarrow_{T_1} [2p_2]_2\}, m_1 \) is the output membrane. The halting computation in the system \( \Pi \) is same as the patterns generated by the pasting system.

\[ \square \]

**Theorem 2.** \( F_{PS} = PL_2(TPPS - \text{AM}, \alpha, \beta) \) where \( \alpha \) is the object evolution rule and \( \beta \) is the out communication rule.

Proof. Consider a Pasting System \( G = (\Sigma, P, t_0) \). A Tile Pasting \( P \) System with active membranes, having developmental rules in the form of object evolution and out communication rule, to generate the language \( L(G) \) is defined below.

\[ \Pi = (O, H, \mu, (t_0)_2', \Sigma', T_k, R_i, m_1) \]

where \( O \) is an alphabet of labels of the edges of the tiles of \( \Sigma \), \( H = \{1, 2\}, \mu = [1|2|2], (t_0)_2' = t_0, m = 2 \) is the initial degree of the system, \( \Sigma' = \Sigma, T_1 = P, R_1 = \{\phi\}, R_2 = \{[2p_1]_2 \rightarrow_{T_1} [2p_2]_2, [2p_1]_2 \rightarrow_{T_1} [2p_2]_2\}, m_1 \) is the output membrane. The halting computation in the system \( \Pi \) is same as the patterns generated by the pasting system.

\[ \square \]

**Theorem 3.** \( F_{PS} \subseteq PL_3(TPPS - \text{AM}, \alpha, \beta) \) where \( \alpha \) is the object evolution rule and \( \beta \) is the object evolution and communication / creation rule.

Proof. Consider a Pasting System \( G = (\Sigma, P, t_0) \). A Tile Pasting \( P \) System with active membranes, having developmental rules in the form of object evolution rule and object evolution & communication / creation rule, to generate the language \( L(G) \) is defined below.

\[ \Pi = (O, H, \mu, (t_0)_2', \Sigma', T_k, R_i, m_1) \]

where \( O \) is an alphabet of labels of the edges of the tiles of \( \Sigma \), \( H = \{1, 2\}, \mu = [2|2|1], (t_0)_2' = t_0, m = 1 \) is the initial degree of the system, \( \Sigma' = \Sigma, T_1 = P, R_1 = \{\phi\}, R_2 = \{[2p_1]_2 \rightarrow_{T_1} [2p_2]_2, [2p_1]_2 \rightarrow_{T_1} [1p_2]_1\}, m_1 \) is the output membrane. The halting computation in the system \( \Pi \) is same as the patterns generated by the pasting system. Hence \( PL_3(\Pi) = L(G) \).

Also the language pattern given in Example 2 can be generated by the following TPPS-AM but not by the PS.

\[ \Pi = (O, H, \mu, (t_0)_1', \Sigma, T_k, R_i, m_3) \]

where \( O = \{R, B\}, H = \{1, 2, 3\}, \mu = [1|2|2|3], (t_0)_2' = t_0, \Sigma \) is the set of tiles considered in Example 2 and \( T_1 = \{(R, B)\}, R_1 = \{[1p_1]_1 \rightarrow_{T_1} [2p_2]_2\}, R_2 = \{[2p_1]_2 \rightarrow_{T_1} [1p_2]_1, [2p_1]_2 \rightarrow_{T_1} [3p_2]_3\}, R_3 = \phi. \)
The halting computation of the system $H$ is the set of patterns containing blue triangles in the outer layer of the pattern. However $PL_3(\text{TPPS} - \text{AM})$ cannot be generated by a pasting system as the language generated by a PS will contain all patterns generated from the axiom tile.

**Theorem 4.** $PL_2(\text{TPPS} - \text{AM}, \alpha, \beta) - F_{\text{EPS}} \neq \emptyset$ where $\alpha$ is the object evolution rule and $\beta \in \{\text{object evolution and dissolution rule, out communication rule}\}$.

**Proof.** Consider the Tile Pasting $P$ system with two membranes having developmental rules in the form of object evolution rule and object evolution and dissolution rule, to generate the tiling pattern using Wang tiles.

$$H = (O, H, \mu, (t_0)_2', \Sigma, T_k, R_1, m_2)$$

where $O = \{R, B, G, Y\}$, $H = \{1, 2\}$, $\mu = [2[1]]_2$, $(t_0)_2' = t_0$, $\Sigma$ is the set of tiles considered in Example 1 and $T_1 = \{(R, R), (B, B), (G, G), (Y, Y)\}$, $R_1 = \{[i|p_1] \rightarrow r_1, [i|p_2] \rightarrow r_2, [i|p_3] \rightarrow r_3\}$, $R_2 = \emptyset$.

The halting computation of the system $H$ is same as the patterns generated in Example 1. In the generation of tessellation patterns using Wang tiles, the edge label set of the tiling pattern could not be generalized as the pattern is a non periodic one. The tessellation pattern generated using Wang tiles cannot be generated using extended pasting scheme, as it is not possible to define the constraint for the edge label set of the tiling.

In a similar manner we can define TPPS-AM having developmental rules in the form of object evolution rule and in the form of out communication rule to generate the Wang tiling.

**Theorem 5.** $F_{\text{TPS}} - PL(\text{TPPS} - \text{AM}, \alpha, \beta) \neq \emptyset$ where $\alpha$ is the object evolution rule and $\beta \in \{\text{object evolution and dissolution rule, out communication rule}\}$.

**Proof.** Consider a tabled pasting system $G = (\Sigma, P, t_0, C)$, where $\Sigma$ is a finite non empty set of tiles; $P$ is a finite set of tables $\{t_1, t_2, \ldots, t_m\}$; each table $t_i$ contains pasting rules to glue the tiles; $t_0$ is the axiom pattern and $C = \{(t_1t_2\ldots t_n)^k/n \geq 3, k = 1, 2, \ldots\}$. For $k = 1$, the tiling pattern is derived in $n$ steps. In step 1, starting from the axiom tile, applying the pasting rules given in the set $t_1$ to the edges of the axiom tile in a maximally parallel manner, we obtain the tiling pattern $p_1$. In a similar manner, a sequence of tiling pattern $p_i$, is obtained for the application of pasting rules from the table $t_2, \ldots, t_n$. Let $p_i^n$ denote the tiling pattern derived by applying the sequence of tables $t_1, t_2, \ldots, t_n$, $i$ number of times. The tiling pattern $p_1^n$ is the pattern computed for $k = 1$. For $k = 2$, we first compute $p_1^n$, as explained above and again apply the sequence of tables $t_1, t_2, \ldots, t_n$, to the pattern $p_1^n$, to obtain the tiling pattern $p_2^n$.

This particular order of, repeated application of tables $t_1, t_2, \ldots, t_n$ to the successive accepted tiling patterns is not possible in Tile Pasting $P$ System with active membranes having developmental rules in the form of object evolution rule, object evolution and dissolution rule and out communication rule, as the
computed tiling pattern $p_n^1$ cannot be communicated back to the membrane $m_1$, containing the table of rules $t_1$, without subjecting $p_n^1$ to other table of rules $t_2, \ldots, t_n$. \hfill \Box

**Theorem 6.** A Tile Pasting $P$ System with active membranes, having developmental rules in the form of object evolution rule $(\alpha)$ and object evolution and communication / creation rule $(\beta)$ is contained in the family of Tabled Pasting System. i.e., $PL(\text{TPPS} - AM, \alpha, \beta) \subset F_{TPS}$.

**Proof.** Consider the tabled pasting system $G = (\Sigma, P, t_0, C)$, where $\Sigma$ is a finite non empty set of tiles; $P = \{t_1, t_2\}$; each table $t_i$ contains pasting rules to glue the tiles; $t_0$ is the axiom tile and $C = \{\{t_1^n t_2^l\}/n = 1, 2, \ldots\}$. A tiling pattern is obtained by applying the pasting rules given in the table $t_1$ $n$ times, followed by the application of pasting rules in $t_2$ $n$ times.

This particular order of applying the table $t_1$ $n$ times followed by the application of table $t_2$ $n$ times is not possible in Tile Pasting $P$ System with active membranes as the system cannot repeat the table $t_2$ as many times it repeated the table $t_1$.

Consider a Tile Pasting $P$ System with active membranes $\Pi = (O, H, \mu, (t_0), \Sigma, T_k, R_i, m_j)$. We define a Tabled Pasting System $G = (\Sigma', P, t_0, C)$ to generate the family of languages $PL_\infty(\text{TPPS} - AM)$. The successive generation of the tiling patterns by the application of developmental rules in the form of object evolution rule and object evolution & communication / creation rule, between the membranes is simulated by arranging the tables $T_k$ according to the developmental rules $R_i$ of the system and defining a control over the tables.

For instance to simulate the developmental set $R_i = \{[p_i]_1 \rightarrow T_k [p_j]_1\}$ containing object evolution rule, we define the repeated application of the table $T_k$, in the control $C$. i.e., $T_k^n, l = 1, 2, \ldots$ in the control $C$. Also to simulate the developmental set $R_i = \{[p_i]_1 \rightarrow T_k [p_j]_1\}$ containing the object evolution and communication / creation rule, we define the application of the table, $T_k$, in the control $C$. Also to simulate the developmental set $R_i = \{[p_i]_1 \rightarrow T_k [p_j]_1, [p_i]_1 \rightarrow T_k [p_j]_1\}$ containing both types of rules (object evolution rule, the object evolution and communication / creation rule), we define the application of the table, $T_k^n T_k, l = 0, 1, \ldots$ in the control $C$. \hfill \Box

**Theorem 7.** $PL(\text{TPPS} - AM, \alpha, \beta) - PL(\Pi) \neq \phi$ where $\alpha$ is the out communication rule and $\beta$ is the object evolution and communication / creation rule.

**Proof.** Consider a Tile Pasting $P$ System with active membranes having developmental rules of the form out communication rule and object evolution and communication / creation rule, $\Pi = (O, H, \mu, (t_0))$, with the membrane structure $\mu = [1][2][3] \ldots [n][n+1][n+1]$, tables $T_1, \ldots, T_n$, and the evolution sets $R_1 = \phi$, $R_2 = \{[2p_i]_2 \rightarrow T_k [3p_j]_3\}$, $\ldots$, $R_n = \{[n+1]_n \rightarrow T_{n-1} [n+2]_{n+1}\}, R_{n+1} = \{[n+1]_{n+1} \rightarrow T_{n+1} [1p_j]_{n+1} \rightarrow T_n [n]_{n+1} \rightarrow T_{n+1} [n+1]_{n+1} p_j\}$.

Starting from the membrane $m_2$, which is having the axiom tile $t_0$, applying the developmental rule given in the set $R_2$ to the edges of the axiom tile in a maximally parallel manner, the resultant tiling pattern $p_1$ is communicated to
In a similar manner, a sequence of tiling pattern $p_i$, $1 < i \leq n$ is communicated to the successive membranes, obtained by applying the developmental rules from the set $R_i$.

The membrane $m_{n+1}$ has two developmental rules, an out communication rule and object evolution and communication rule. During the pattern generation any one of them is non-deterministically chosen and applied to the tiling pattern. If the out communication rule is chosen then the resultant pattern is communicated to the membrane $m_1$ and the computation stops as $m_1$ is the output membrane. If the evolution and communication is chosen then the computed pattern is communicated back to the membrane $m_2$, resulting in the application of the developmental sets $R_2, \ldots, R_n, R_{n+1}$ once again in a sequential manner.

This particular way of, repeated application of developmental sets $R_2, \ldots, R_n, R_{n+1}$ to the successive computed tiling patterns is not possible in Tile Pasting $P$ System as the computed tiling pattern cannot be communicated back to the membrane $m_2$, containing the developmental rule $R_2$, without subjecting the computed pattern to other developmental set of rules $R_3, \ldots, R_n, R_{n+1}$. □

5 Conclusion

Tile Pasting $P$ Systems with active membranes (TPPS-AM) is a variant of Tile Pasting $P$ System, based on the structure and functioning of living cells to generate tiling patterns using pasting rules. In this paper the efficiency of TPPS-AM, with different combination of developmental rules, is discussed and the system is compared with other models such as PS, EPS, TPS and TPPS.

References

Matrix Representation of Spiking Neural P Systems with Delay

Kamala Krithivasan$^1$ and Ajeesh Ramanujan$^1$

$^1$Department of Computer Science and Engineering
Indian Institute of Technology Madras, Chennai - 36
kamala@iitm.ac.in, ajeeshramanujan@yahoo.com

Abstract. We consider a method of representing a spiking neural P systems using matrices and vectors, so that successive configurations of the system can be obtained by a procedure involving proper operations on these matrices and vectors. The idea is as well extended to spiking neural P systems with anti-spikes.

1 Introduction

Membrane computing is a new model of computation initiated by Păun [2]. This computational model is inspired by the structure and the functioning of the living cells and the ways in which cells are organized in tissues and higher order biological structures. This model of computing is called P systems. P systems are distributed and parallel models, which process multisets of objects in the compartments defined by membranes. Refer [6] for the current developments in the area of membrane computing.

Soon after the introduction of P systems, different variants were proposed. Most variants of membrane systems have been proved to be computationally complete. Some variants are useful for solving computationally hard problems in polynomial time [6, 3].

In this paper we consider a such variant called spiking neural P system (shortly called SN P system). This variant was introduced in [1] and is a biologically inspired parallel and distributed computing model motivated by neurobiological behavior of neurons sending electrical pulses of identical voltages called spikes to neighboring neurons. We also consider a variant of SN P system called spiking neural P system with anti-spikes (shortly called SN PA system). This variant was introduced in [5].

An SN P system consists of a set of processing elements called neurons placed in the nodes of a directed graph called synapse graph with each neuron having a number of copies of a single type object called spikes. Each neuron may contain rules that allow to remove a specified number of spikes from a neuron or to transfer spikes to the neighboring neurons. It is also possible to introduce a delay in transferring spikes between neurons. The application of a rule in a neuron is determined by checking the number of spikes in the neuron with a regular expression.
An SN PA system is similar to an SN P system, but in an SN PA system we use two types of objects called spikes and anti-spikes. But only one type of object, either spike or anti-spike exists in a neuron at a particular instance. If they happen to come together in a neuron they annihilate each other.

In this paper we consider a discrete representation of spiking neural P systems with delay and spiking neural P systems with anti-spikes using matrices. One such discrete representation of spiking neural P systems without delay using matrices was proposed by Zeng et al. [4] and with delay is left as an open problem in [4].

The rest of the paper is organized as follows. In Section 2 we give some definitions and notations related to automata theory and linear algebra. In Section 3 we define the standard version of SN P systems (as defined in [1]) and SN PA systems in Section 4. In Section 5 we define the vectors, matrices and operators used in this paper. In Section 6 a procedure is explained how to perform computation using the vectors, matrices and operators and we prove the correctness of the procedure. In Section 7 a matrix representation of SN PA system is explained. Section 8 concludes the paper with some future directions.

2 Basic Definition

In this section we give some definitions and notations related to automata theory and linear algebra.

Let \( \Sigma \) be a finite set of symbols called an alphabet. A string \( w \) over \( \Sigma \) is a sequence of symbols from \( \Sigma \). \( \lambda \) denotes the empty string. The set of all strings over \( \Sigma \) is denoted by \( \Sigma^* \). A language \( L \) over \( \Sigma \) is a set of strings over \( \Sigma \).

A language \( L \subseteq \Sigma^* \) is said to be regular if there is a regular expression \( E \) over \( \Sigma \) such that \( L(E) = L \). The regular expressions are defined using the following rules. (i.) \( \emptyset, \lambda \) and each \( a \in \Sigma \) are regular expressions. (ii.) If \( E_1 \) and \( E_2 \) are regular expressions over \( \Sigma \), then \( E_1 + E_2, E_1 E_2 \) and \( E_1^* \) are regular expressions over \( \Sigma \) and (iii.) nothing else is a regular expression over \( \Sigma \). With each regular expression \( E \), we associate a language \( L(E) \).

In order to represent the computations of SN P systems we use row vectors and matrices. A row vector is written in boldface letter. For example \( u_i \) represents a row vector \( u \). In this paper all the vectors are subscripted. The \( j \)th component of a vector \( u_i \) is denoted by \( u_{ij} \). Matrices are represented by capital letters. For a matrix \( M \), the \( ij \)th entry is denoted by \( M_{ij} \).

3 Spiking Neural P (SN P) Systems with Delay

In this section we give the basic definition of spiking neural P system with delay.

**Definition 1.** A spiking neural P system of degree \( m \geq 1 \) with delay is a construct of the form \( \Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, i_0) \), where

1. \( O = \{a\} \) is a singleton alphabet (\( a \) is called spike);
2. $\sigma_1, \sigma_2, \cdots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where
   (a) $n_i \geq 0$ is the initial number of spikes contained in the neuron $\sigma_i$;
   (b) $R_i$ is a finite set of rules of the form
      
      i. $E/a^r \rightarrow a; t$, where $E$ is a regular expression over $O, r \geq 1$, and $t \geq 0$; The number of spikes present in the neuron is described by the regular expression $E$, $r$ spikes are consumed and it produces a spike, which will be send to other neurons after $t$ steps;
      
      ii. $a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule $E/a^r \rightarrow a; t, t \geq 0$.

3. $\text{syn} \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\}$ with $(i, i) \notin \text{syn}$ for $1 \leq i \leq m$;

4. $i_0 \in \{1, 2, \cdots, m\}$ indicates the output neuron.

The rules of type $E/a^r \rightarrow a; t$, are spiking rules, and they are applicable only if the neuron contains $n$ spikes such that $a^n \in L(E)$ and $n \geq r$. If $E = \emptyset$ then the rule is applied only if the neuron contains exactly $r$ spikes. When neuron $\sigma_i$ spikes, $r$ spikes are consumed and its spike is replicated in such a way that one spike is sent to all neurons $\sigma_j$ after $t$ time units (a global clock is assumed that synchronizes the functioning of the system) such that $(i, j) \in \text{syn}$, and $\sigma_j$ is open at that moment. If a neuron $\sigma_i$ fires and it has no outgoing synapse, or all neurons $\sigma_j$ such that $(i, j) \in \text{syn}$ are closed, then the spike of $\sigma_i$ is lost. So if a rule $a^s \rightarrow \lambda$ is used in neuron, say $\sigma_j$, in the $k$th step then the neuron $\sigma_j$ will be closed for the next $t$ steps, and can send the spikes to other neurons connected to it in the $(k+t)$th step. During this time the neuron $\sigma_j$ will not be able to receive spikes from other neurons or not able to use any rules. The neuron $\sigma_j$ can start using rules only from the $(k+t)$th step onwards.

The rules of type $a^s \rightarrow \lambda$ are forgetting rules. $s$ spikes are removed when applied. Like in the case of spiking rules, the left hand side of a forgetting rule must cover the contents of the neuron, that is, $a^s \rightarrow \lambda$ is applied only if the neuron contains exactly $s$ spikes.

Each neuron which can use a rule should do it. The working of the system is locally sequential, at most one rule can be used during a computation step. One of the neuron is considered to be the output neuron, and its spikes are sent to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1; the other moments are marked with 0. This binary sequence is called the spike train of the system. It might be infinite if the computation does not stop. Many kinds of output can be associated with a computation in an SN P system. In this paper we consider the distance between the first two spikes in the spike train as the value computed by an SN P system.

We give an example from [1] to illustrate the above definition.

3.1 An Example

Consider the pictorial representation of an SN P system given in Figure 1. Each neuron is represented by an oval and it is marked with a label. The current number of spikes and the evolution rules are placed inside the oval. The synapses between the neurons are represented by directed arrows. The output neuron is
either identified by its label, $i_0$ or shown by drawing an arrow pointing to the environment.

![Diagram](https://via.placeholder.com/150)

*Fig. 1. An example of an SN P system.*

In the SN P system given in Figure 1, we have three neurons with labels 1, 2 and 3. Neuron 3 is the output neuron. Neurons 1 and 3 has spikes initially. So they can fire in the first step. The spike of neuron 3 exits the system, so the number of steps from now until the next spiking of neuron 3 is the number computed by the system. Neuron 3 does not have any more spike to fire and it will remain inactive until a new spike comes in. Neuron 2 can spike only after collecting $k$ spikes.

After step one, neuron 1 will be inactive for the next two steps. In the third step it will release its spike, sending it to neuron 2, and in step 3 will fire again. Neuron 1 fires in every third step, consuming one of the spikes. In the $3k$th step neuron 2 will receive the $k$th spike emitted by neuron 1. In the $3k + 1$th step neuron 1 will fire. The delay between the firing and spiking is of one time unit for neuron 2, hence its spike will reach neuron 3 in step $3k + 2$. The neuron 3 spikes again in step $3k + 3$. So, the computed number is $(3k + 3) - 1 = 3k + 2$. The computation continues consuming spikes in neuron 1, until the $k - 1$ spikes are passed on to neuron 2 when the computation stops.

4 Spiking Neural P Systems with Anti-Spikes

In this section we give the basic definition of SN PA system.

**Definition 2.** An SN PA system of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \sigma_2, \cdots, \sigma_m, \text{syn}, i_0)$, where

1. $O = \{a, \bar{a}\}$ is the alphabet. $a$ is called spike and $\bar{a}$ is called anti-spike;
2. $\sigma_1, \sigma_2, \cdots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, where
   (a) $n_i \geq 0$ is the initial number of spikes or anti-spikes contained in the neuron $\sigma_i$;
   (b) $R_i$ is a finite set of rules of the form
   i. $E/b' \rightarrow b';t$, where $E$ is a regular expression over $O, b \in O, r \geq 1$ and $t \geq 0$; Number of spikes or anti-spikes present in the neuron is described by the regular expression $E, r$ spikes or anti-spikes are consumed and it produces a spike or anti-spike, which will be sent to other neurons after $t$ steps;
ii. \( b^s \rightarrow \lambda \), for some \( s \geq 1 \), with the restriction that \( b^s \not\in L(E) \) for any rule \( E/b^r \rightarrow b/t \), of type (i) from \( R_i \):

3. \( \text{syn} \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\} \) with \((i, i) \not\in \text{syn} \) for \( 1 \leq i \leq m \);

4. \( i_0 \in \{1, 2, \cdots, m\} \) indicates the output neuron.

The working of SN PA system is similar to that of SN P system. But in the case of SN PA system, the spikes and the anti-spikes cannot stay together. They annihilate each other. If a neuron has either objects \( a \) or \( \bar{a} \), and further objects of either type arrive from other neurons, such that we end with \( a^r \) and \( \bar{a}^s \) inside a neuron, then immediately an annihilation rule \( a^r \bar{a}^s \rightarrow \lambda \), which is implicit in each neuron is applied in a maximal manner, so that either \( a^r - s \) or \( \bar{a}^s - r \) remains for the next step, provided that \( r \geq s \) or \( s \geq r \). This mutual annihilation of spikes and anti-spikes takes no time and that annihilation rule has priority over spiking and forgetting rules. So the neurons always contain either only spikes or anti-spikes.

5 Vectors, Matrices and Operators

In this section we define the vectors, matrices and operators used in this paper.

Consider an SN P system \( \Pi \) of degree \( m \). Let the the number of rules be \( n \). Assume an ordering \( d_1, d_2, \cdots, d_n \) on the rules in such a way that the rules in the same neuron are given consecutive labels and the neurons are considered in the increasing order of their labels. For a rule \( d_i : a^r \rightarrow a^s ; t \), we define \( \text{lhs}(d_i) = r, \text{rhs}(d_i) = s \) and \( \text{delay}(d_i) = t \).

The configuration vector is described by the number of spikes present in each neuron.

**Definition 3 (Configuration vector).** The vector \( \mathbf{c}_i = (n_{i_1}, n_{i_2}, \cdots, n_{i_m}) \), where \( n_{i_j} \geq 0, 1 \leq j \leq m \) and \( n_{i_1}, n_{i_2}, \cdots, n_{i_m} \) are the number of spikes contained in the \( m \) neurons at step \( i \), is called the configuration vector of the system at step \( i \).

A spiking vector is used to describe which rules are selected and applied at step \( i \).

**Definition 4 (Spiking vector).** The vector \( \mathbf{s}_i \in \{0, 1\}^n \) used at step \( i \) is defined as follows

\[
\mathbf{s}_i = \begin{cases} 
1 & \text{if the regular expression corresponding to rule } d_j \text{ is satisfied and the rule } d_j \text{ is chosen and applied;} \\
0 & \text{otherwise.}
\end{cases}
\]

\( 1 \leq j \leq n \).

This is called the spiking vector at step \( i \).

A delay vector is used to describe which neurons are inactive at step \( i \).

**Definition 5 (Delay vector).** The vector \( \mathbf{d}_i = (n_{i_1}, n_{i_2}, \cdots, n_{i_m}) \), where \( n_{i_j} \geq 0, 1 \leq j \leq m \) and if \( n_{i_j} > 0 \) then neuron \( j, 1 \leq j \leq m \) is inactive at step \( i \), is called the delay vector at step \( i \).
A transfer vector is used to transfer spikes to neurons whose state changed from inactive to active state at step $i$.

**Definition 6 (Transfer vector).** The vector $tv_i = (n_{i_1}, n_{i_2}, \ldots, n_{i_n})$, where $n_{i_j} \geq 0, 1 \leq j \leq n$ and if $n_{i_j} = 1$ then the spike corresponding to rule $d_j, 1 \leq j \leq m$ can be sent to the neurons connected to the neuron containing the rule $d_j, 1 \leq j \leq m$ is called the transfer vector at step $i$.

A collection of zero vectors equal to the number of neurons called rule vectors are used to set spiking vector during computation. This helps to find whether a set of rules is applicable at a particular instance or not.

**Definition 7.** A collection of zero vectors $r_1, r_2, \ldots, r_m$ where the size of $r_i$ is equal to the number of rules in neuron $i$ are called the rule vectors.

Next we define the matrices used to transfer spikes among neurons and used for manipulating vectors defined above.

When the rules are chosen and applied at every step, the reduction in the number of spikes in different neurons is represented by consumption matrix.

**Definition 8 (Consumption Matrix).** The matrix $C$ of size $n \times m$ whose entities defined as follows

$$
c_{ij} = \begin{cases} 
-\text{lhs}(d_i) & \text{if there is a rule } d_i : a^r \rightarrow a^s; t \text{ in neuron } \sigma_j; \\
0 & \text{otherwise.}
\end{cases}
$$

is called the consumption matrix.

When the rules are chosen and applied at every step, the gain in the number of spikes in different neurons is represented by transfer matrix.

**Definition 9 (Transfer Matrix).** The matrix $T$ of size $n \times m$ whose entities defined as follows

$$
t_{ij} = \begin{cases} 
\text{rhs}(d_i) & \text{if there is a rule } d_i : a^r \rightarrow a^s; t \text{ in neuron } \sigma_s \text{ and } s \neq j, (s, j) \in \text{syn}; \\
0 & \text{otherwise.}
\end{cases}
$$

is called the transfer matrix.

The delay induced by the rules in the neurons is represented by delay matrix.

**Definition 10 (Delay Matrix).** The matrix $D$ of size $n \times m$ whose entities defined as follows

$$
d_{ij} = \begin{cases} 
\text{delay}(d_i) + 1 & \text{if there is a rule } d_i : a^r \rightarrow a^s; t \text{ in neuron } \sigma_j \text{ and } t > 0; \\
0 & \text{otherwise.}
\end{cases}
$$

is called the delay matrix.

In order to control the transfer of spikes from the neurons when a neuron comes out of its inactive state a firing matrix is used.
Definition 11 (Firing Matrix). The matrix $F$ of size $n \times m$ whose entities defined as follows

$$f_{ij} = \begin{cases} 
\text{delay}(d_i) + 1 & \text{if there is a rule $d_i : a^r \rightarrow a^s$; $t$ in neuron } \sigma_j \text{ and if } \sigma_j \in i_0 \\
0 & \text{otherwise.}
\end{cases}$$

is called the firing matrix.

Next we define the operators that are used for manipulating the different vectors and matrices.

Operator 1 $\otimes$
For $u, v \in \mathbb{Z}^n$.

$$u \otimes v = \begin{cases} 
0 & \text{if } u_i > 0; \\
v_i & \text{otherwise.}
\end{cases}$$

Operator 2 $\odot$
For $u \in \mathbb{Z}^n, v \in \mathbb{Z}^m$ $u \odot v = (u_1, u_2, \cdots, u_n, v_1, v_2, \cdots, v_m)$.

Operator 3 dec
For $v \in \mathbb{Z}^n$.

$$\text{dec}(v) = \begin{cases} 
v_i - 1 & \text{if } v_i > 0; \\
0 & \text{otherwise.}
\end{cases}$$

Operator 4 $\leftarrow$
For $u, v \in \mathbb{Z}^n$.

$$u \leftarrow v = \begin{cases} 
1 & \text{if } v_i = 1; \\
0 & \text{otherwise.}
\end{cases}$$

Operator 5 $\oplus$
For $u, v \in \mathbb{Z}^n$.

$$u \oplus v = \begin{cases} 
v_i & \text{if } u_i = 1; \\
u_i + v_i & \text{otherwise.}
\end{cases}$$

Operator 6 $\triangleleft \triangleright$
For $v \in \mathbb{Z}^n, c \in I^+$.

$$c \triangleleft \triangleright v = \begin{cases} 
0 & \text{if } c > 0; \\
1 & \text{otherwise.}
\end{cases}$$

Operator 7 $\prod$
For $u, v \in \mathbb{Z}^n$.

$$u \prod v = \begin{cases} 
1 & \text{if } u_i = 1 \text{ and } v_i = 1; \\
0 & \text{otherwise.}
\end{cases}$$

Operator 8 $\times$
For $u \in \mathbb{Z}^n, M^{n \times m} \in \mathbb{Z}$.

$$u \times M = u_i \left(\sum_{j=1}^{m} m_{ij}\right).$$
6 Computation by Matrices

In this section, a procedure is explained how to perform computation using the vectors, matrices and operators defined in the above section.

Consider an SN P system \( \Pi \) of degree \( m \). Let the the number of rules be \( n \). Assume an ordering \( d_1, d_2, \cdots, d_n \) on the rules in such a way that the rules within a neuron are given consecutive labels and the neurons are considered in the increasing order of their labels. Let \( c_0 \) be the initial configuration vector. The initial delay vector \( dv_0 \) and the transfer vector \( tv_0 \) are initialized to zero.

The matrices are defined as per the definition given in section 5. We use two temporary vectors \( s' \) and \( tv' \) for computation. The \( i \)th configuration vector, \( c_i, i \geq 1 \) is found out by executing the following steps.

Step 1 \( dec(dv_{i-1}) \); decrement the delay vector using operator \( dec \).
Step 2 \( dec(tv_{i-1}) \); decrement the transfer vector using operator \( dec \).
Step 3 Find a spiking vector, \( s_i \). The spiking vector is either provided by the user or by an automated system that checks the regular expression associated with the rule against the number of spikes in each neuron.
Step 4 \( s' = (dv_{i_1} \bowtie r_1) \odot (dv_{i_2} \bowtie r_2) \odot \cdots \odot (dv_{i_m} \bowtie r_m) \); \( s' \) is used to forbid the action of the rules within a neuron that is in the inactive state. For example if neuron \( i \) is inactive and suppose that it contains say \( k \) rules with labels \( d_j, d_{j+1}, \cdots, d_{j+k-1} \) then this operation sets the components of \( s' \) from \( j, j+1, \cdots, j+k-1 \) to zero.
Step 5 \( s_i = s_i \prod s' \); sets \( s_i \) as per the result of step 4 thereby forbidding the action of rules within an inactive neuron.
Step 6 \( c_i = c_{i-1} + s_i \cdot C \); update the configuration vector by taking away the spikes from neurons if possible.
Step 7 \( tv' \leftarrow tv_i \); set the component, say \( j \) of the temporary vector if a spike can be transferred after the delay caused by the rule \( d_j \) has expired.
Step 8 \( c_i = c_i + (dv_i \odot (tv'.T)) \); update the configuration vector by adding spikes if possible.
Step 9 \( tv_i = tv_i \oplus (s_i \times F) \); update the transfer vector.
Step 10 \( dv_i = dv_i + s_i \cdot D \); update the delay vector.
Step 11 \( tv' \leftarrow tv_i \)
Step 12 \( c_i = c_i + (dv_i \odot (tv'.T)) \); steps 11 and 12 are similar to steps 7 and 8 and are used to update the configuration vector if a zero delay rule is used.

We demonstrate the working of the procedure with an example. Consider the SN P system given in section 3.1. We run the procedure for \( k = 2 \). The different matrices are defined as follows.

\[
C = \begin{bmatrix}
-1 & 0 & 0 \\
0 & -2 & 0 \\
0 & 0 & -1
\end{bmatrix},
T = \begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{bmatrix},
D = \begin{bmatrix}
3 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 0
\end{bmatrix},
F = \begin{bmatrix}
3 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]
The rule vectors are $r_1 = (0), r_2 = (0), r_3 = (0)$. The working of the procedure is explained in Appendix A.

The nondeterminism is handled as follows. Suppose that at a point during computation more than one spiking vector is applicable. Then the algorithm creates its own copies that is equal to the number of spiking vectors that are applicable at that point and the computation proceeds by branching. Each branch corresponds to a copy. Each copy shares the consumption matrix, transfer matrix, delay matrix and the firing matrix. But each copy has its own copy of configuration vector, spiking vector, delay vector and transfer vector. Each copy continues its computation in parallel. The result of the computation is the set of values computed by the branches that halts.

In the remaining part of this section we prove the correctness of the above described procedure.

In the following lemmas we consider an SN P system $\Pi$ with $m$ neurons and $n$ rules. Assume an ordering $d_1, d_2, \cdots, d_n$ on the rules in such a way that the rules within a neuron are given consecutive labels and the neurons are considered in the increasing order of their labels.

**Lemma 1.** Let $l_i, 1 \leq i \leq m$, $\sum_{i=1}^{m} l_i = n$ be the number of rules in neuron $i$. In the $k$th step of computation, the vector $s^\prime$ contains information about which rules can be applied.

**Proof.** Consider a particular neuron, say $\sigma_j$. The number of rules in $\sigma_j$ is $l_j$. If $\sigma_j$ is inactive, then $d_{k_j} > 0$. So the operation $dv_{k_j} \otimes r_1$ sets components $i, i+1, \cdots, i+l_j-1$, where $i = \sum_{q=1}^{j-1} l_q$ to 0. Else $d_{k_j} = 0$ and the components $i, i+1, \cdots, i+l_j-1$, where $i = \sum_{q=1}^{j} l_q$ are set to 1. The same operation is done with other neurons and the vector $s^\prime$ is built by performing the operation $(dv_{k_1} \otimes r_1) \odot (dv_{k_2} \otimes r_2) \odot \cdots \odot (dv_{k_m} \otimes r_m)$ in step 4. If a component of $s^\prime$, say $i$ is equal to 1, the rule $d_i$ can be applied in the $k$th step. Therefore, the vector $s^\prime$ contains information about which rules can be applied.

**Lemma 2.** If the $j$th component of $s_k$ obtained in step 5 of the procedure in the $k$th step of computation is 1, then the rule with label $d_j$ can be used.

**Proof.** By Lemma 1, we know that the temporary vector $s^\prime$ is properly set. The operation $s_k = s_k \prod i s^\prime$ sets the $j$th component of $s_k$ to 1 if $s_{k_j} = 1$ and $s_{j}^\prime = 1$. So the neuron containing the rule $d_j$ is active and the rule $d_j$ can be used in the $k$th step.

**Lemma 3.** If there is a rule $d_i : a^p \rightarrow a;t$ in neuron $j$ and neuron $j$ is active, then the application of the rule decrements the number of spikes in neuron $j$ by $p$.

**Proof.** Suppose that the neuron $j$ is active and there is a rule $d_i : a^p \rightarrow a;t$ in neuron $j$. Suppose that the system is in the $k$th step of the computation and the rule $d_i$ is selected. By Lemma 2, we know that the spiking vector $s_k$ is properly set. So the $i$th component of $s_k$ is set to 1. The operation $s_k, C$ returns the
Lemma 4. Let \( l_i, 1 \leq i \leq m, \sum_{i=1}^{n} l_i = n \) be the number of rules in neuron \( i \). In the \( k \)th step of computation, the vector \( tv' \) contains information about whether the spikes can be transferred or not.

Proof. The operation \( tv' \leftarrow tv_k \) sets a component of \( tv' \) to 1 if the corresponding component of \( tv_k \) is 1 otherwise it will be set to 0. If the \( j \)th component of \( tv_k \) is 1 then it means that the rule \( d_j \) is used in some previous step and if possible the spike can be transferred to the active neurons connected to the neuron containing the rule \( d_j \). So the vector \( tv' \) contains information about whether the spikes can be transferred or not.

Lemma 5. In the \( k \)th step of computation, if a neuron, say \( \sigma_j \), is active and if a rule \( d_k : a^p \rightarrow a; t \) is applied in neuron \( \sigma_i \) at the \((k-t)\)th step and if \((l,j) \in \text{syn} \) then \( \text{rhs}(d_i) \) number of spikes are transferred to neuron \( \sigma_j \).

Proof. By Lemma 4, the temporary vector \( tv' \) will be properly set. Consider the \( \text{ith} \) component of \( tv' \). Let it be 1. Then the operation \( tv'_i.T \) returns \( T_{ij}, i.e. \text{rhs}(d_i) \) if the rule \( d_i : a^p \rightarrow a; t \) is applied in neuron \( \sigma_i \) at the \((k-t)\)th step of computation and \((l,j) \in \text{syn} \) and \( \sigma_j \) is active in the \( k \)th step of computation. Since \( \sigma_j \) is active, \( dv_{kj} = 0 \). So the operation \( dv_{kj} \otimes (tv'_i.T) \) return the result of the operation \( tv'_i.T \), i.e. \( \text{rhs}(d_i) \). So the \( j \)th component of the configuration vector is updated after performing the operation \( c_{kj} = c_{kj} + (dv_{kj} \otimes (tv'_i.T)) = c_{kj} + \text{rhs}(d_i) \).

Lemma 6. In the \( k \)th step of computation, if a rule \( d_k : a^p \rightarrow a; t \) is used in neuron \( l \), then the \( \text{ith} \) component of the transfer vector, \( tv_{ki} \), will be set to \( \text{delay}(d_i) + 1 \).

Proof. Suppose that a rule \( d_k : a^p \rightarrow a; t \) is used in neuron \( l \) in the \( k \)th step of the computation. By Lemma 2, the \( i \)th component of \( s_k \) will be 1. So the operation \( s_{ki} \times F \) returns \( F_{il} \) which is equal to \( \text{delay}(d_i) + 1 \). Therefore \( tv_{ki} \) will be set to \( \text{delay}(d_i) + 1 \) after performing the operation \( tv_{ki} = tv_{ki} \oplus (s_{ki} \times F) \).

Lemma 7. In the \( k \)th step of computation, if a rule \( d_k : a^p \rightarrow a; t \) is used in neuron \( l \) and \( \text{delay}(d_i) > 0 \), then the \( \text{ith} \) component of the delay vector, \( dv_{ki} \), will be set to \( \text{delay}(d_i) + 1 \) if \( \text{delay}(d_i) > 0 \).

Proof. Suppose that a rule \( d_k : a^p \rightarrow a; t \) is used in neuron \( l \) in the \( k \)th step of the computation. By Lemma 2, the \( i \)th component of \( s_k \) will be 1. So the operation \( s_{ki} \times D \) returns \( D_{il} \) which is equal to \( \text{delay}(d_i) + 1 \) if \( \text{delay}(d_i) > 0 \). Therefore \( dv_{ki} \) will be set to \( \text{delay}(d_i) + 1 \) if \( \text{delay}(d_i) > 0 \) after performing the operation \( dv_{ki} = dv_{ki} + s_{ki} \times D \).

Lemma 8. In the \( k \)th step, if a neuron, say \( \sigma_j \), is active and if a rule \( d_k : a^p \rightarrow a; 0 \) is applied in neuron \( \sigma_i \) and if \((l,j) \in \text{syn} \) then \( \text{rhs}(d_i) \) number of spikes are transferred to neuron \( \sigma_j \).
Proof. The proof is similar to the proof of Lemma 5. But here we use the updated transfer vector and the delay vector in the Step 9 and step 10 of the procedure that are updated during the \( k \)th step of computation.

**Theorem 1.** In the \( k \)th step of computation, the \( k \)th configuration vector, \( c_k \) will be correctly computed from the \( (k-1) \)th configuration vector \( c_{k-1} \) by the procedure.

Proof. The correctness of the procedure follows from Lemmas 1 through 8.

### 7 Matrix representation of SN PA systems

In this section we define the vectors and matrices for representing SN PA system that are different from the definition given in section 5.

In the case of SN PA system, the configuration vector is described by the number of spikes or anti-spikes present in each neuron.

**Definition 12 (Configuration vector).** The vector \( c_i = (n_{i1}, n_{i2}, \ldots, n_{im}) \), where \( n_{ij} \in \mathbb{Z}, 1 \leq j \leq m \) and \( n_{i1}, n_{i2}, \ldots, n_{im} \) are the number of spikes or anti-spikes contained in the \( m \) neurons at step \( i \), is called the configuration vector of the system at step \( i \). Here, \( n_{ij}, 1 \leq j \leq m \) is positive if neuron \( \sigma_j \) contains spikes, else if \( n_{ij}, 1 \leq j \leq m \) is negative the neuron \( \sigma_j \) contains anti-spikes.

When the rules are chosen and applied at every step, the reduction in the number of spikes or anti-spikes in different neurons is represented by consumption matrix.

**Definition 13 (Consumption Matrix).** The matrix \( C \) of size \( n \times m \) whose entities defined as follows

\[
c_{ij} = \begin{cases} 
-lhs(d_i) & \text{if there is a rule } d_i : a^r \rightarrow b^s; t, b \in \{a, \bar{a}\} \text{ in neuron } \sigma_j; \\
-lhs(d_i) & \text{if there is a rule } d_i : \bar{a}^r \rightarrow b^s; t, b \in \{a, \bar{a}\} \text{ in neuron } \sigma_j; \\
0 & \text{otherwise.}
\end{cases}
\]

is called the consumption matrix.

When the rules are chosen and applied at every step, the gain in the number of spikes or anti-spikes in different neurons is represented by transfer matrix.

**Definition 14 (Transfer Matrix).** The matrix \( T \) of size \( n \times m \) whose entities defined as follows

\[
t_{ij} = \begin{cases} 
\text{rhs}(d_i) & \text{if there is a rule } d_i : b^r \rightarrow a^s; t, b \in \{a, \bar{a}\} \text{ in neuron } \sigma_s \\
\text{and } s \neq j, (s, j) \in \text{syn}; \\
-\text{rhs}(d_i) & \text{if there is a rule } d_i : \bar{a}^r \rightarrow b^s; t, b \in \{a, \bar{a}\} \text{ in neuron } \sigma_s \\
\text{and } s \neq j, (s, j) \in \text{syn}; \\
0 & \text{otherwise.}
\end{cases}
\]

is called the transfer matrix.

The computations using SN PA systems can be done by using the procedure explained in section 6 using the modified configuration vector, consumption matrix and the transfer vector.
8 Conclusion

In this paper, we have considered the method of finding the configuration of an SN P system with delay by using matrices. We have also considered the method of finding the configuration of an SN P system with antispikes by using matrices. Different SN P systems could be simulated with different types of Petri Nets. At present we are exploring the possibility of representing the configuration of a Petri Net at any particular instance using matrices and the correspondence between that procedure and the procedure discussed here.

References


Appendix A

1. Step 0
   (a) \( c_0 = (3, 0, 1), d_v_0 = (0, 0, 0), t_v_0 = (0, 0, 0) \).
2. Step 1
   (a) \( d_v_1 = (0, 0, 0) \)
   (b) \( t_v_1 = (0, 0, 0) \)
   (c) Give \( s_1 = (1, 0, 1) \)
   (d) \( s = (d_v_1 \bowtie r_1) \circ (d_v_2 \bowtie r_2) \circ (d_v_3 \bowtie r_3) = (0 \bowtie (0)) \circ (0 \bowtie (0)) \circ (0 \bowtie (0)) = (1, 1, 1) \)
   (e) \( s_1 = (1, 0, 1) \prod (1, 1, 1) = (1, 0, 1) \)
   (f) \( c_1 = (3, 0, 1) + (1, 0, 1), C = (3, 0, 1) + (-1, 0, -1) = (2, 0, 0) \)
   (g) \( t_v' \leftarrow (0, 0, 0) \) sets \( t_v' \) to \( (0, 0, 0) \)
   (h) \( c_1 = (2, 0, 0) + (0, 0, 0) \otimes ((0, 0, 0), T) = (2, 0, 0) + (0, 0, 0) = (2, 0, 0) \)
   (i) \( t_v_1 = (0, 0, 0) \oplus (1, 0, 1) \times F) = (0, 0, 0) \oplus (3, 0, 0) = (3, 0, 0) \)
   (j) \( d_v_1 = (0, 0, 0) + (1, 0, 1), D = (0, 0, 0) + (3, 0, 0) = (3, 0, 0) \)
   (k) \( t_v' \leftarrow (3, 0, 0) \) sets \( t_v' \) to \( (0, 0, 0) \)
   (l) \( c_1 = (2, 0, 0) + ((3, 0, 0) \otimes ((0, 0, 0), T)) = (2, 0, 0) + ((3, 0, 0) \otimes (0, 0, 0)) = ((2, 0, 0) + (0, 0, 0) = (2, 0, 0) \)
3. Step 2
   (a) \( d_v_2 = (2, 0, 0) \)
   (b) \( t_v_2 = (2, 0, 0) \)
(c) Give $s_2 = (1, 0, 0)$
(d) $s' = (dv_{21} \times r_1) \odot (dv_{22} \times r_2) \odot (dv_{23} \times r_3) = (2 \times (0)) \odot (0 \times (0)) \odot (0 \times (0)) = (0, 1, 1)$
(e) $s_3 = (1, 0, 0) \prod (0, 1, 1) = (0, 0, 0)$
(f) $c_2 = (2, 0, 0) + (0, 0, 0), C = (2, 0, 0) + (0, 0, 0) = (2, 0, 0)$
(g) $tv \leftarrow (2, 0, 0)$ sets $tv$ to $(0, 0, 0)$
(h) $c_2 = (2, 0, 0) + (0, 0, 0) \odot ((0, 0, 0), r) = (2, 0, 0) + (0, 0, 0) = (2, 0, 0)$
(i) $tv_2 = (0, 0, 0) \odot (0, 0, 0) \times F = (0, 0, 0) \odot (2, 0, 0) = (2, 0, 0)$
(j) $dv_2 = (2, 0, 0) + (0, 0, 0), D = (2, 0, 0) + (0, 0, 0) = (2, 0, 0)$
(k) $tv \leftarrow (2, 0, 0)$ sets $tv$ to $(0, 0, 0)$
(l) $c_2 = (2, 0, 0) + ((2, 0, 0) \odot (0, 0, 0)) = (2, 0, 0) + ((2, 0, 0) \odot (0, 0, 0)) = (2, 0, 0) + (0, 0, 0) = (2, 0, 0)$

4. Step 3
(a) $dv_3 = (1, 0, 0)$
(b) $tv_3 = (1, 0, 0)$
(c) Give $s_3 = (1, 0, 0)$
(d) $s' = (dv_{31} \times r_1) \odot (dv_{32} \times r_2) \odot (dv_{33} \times r_3) = (1 \times (0)) \odot (0 \times (0)) \odot (0 \times (0)) = (0, 1, 1)$
(e) $s_3 = (1, 0, 0) \prod (0, 1, 1) = (0, 0, 0)$
(f) $c_3 = (2, 0, 0) + (0, 0, 0), C = (2, 0, 0) + (0, 0, 0) = (2, 0, 0)$
(g) $tv \leftarrow (1, 0, 0)$ sets $tv$ to $(1, 0, 0)$
(h) $c_3 = (2, 0, 0) + (1, 0, 0) \odot ((1, 0, 0), r) = (2, 0, 0) + (0, 1, 0) = (2, 1, 0)$
(i) $tv_3 = (1, 0, 0) \odot (0, 0, 0) \times F = (1, 0, 0) \odot (0, 0, 0) = (0, 0, 0)$
(j) $dv_3 = (1, 0, 0) + (0, 0, 0), D = (1, 0, 0) + (0, 0, 0) = (1, 0, 0)$
(k) $tv \leftarrow (2, 0, 0)$ sets $tv$ to $(0, 0, 0)$
(l) $c_3 = (2, 1, 0) + ((1, 0, 0) \odot (0, 0, 0)) = (2, 1, 0) + ((1, 0, 0) \odot (0, 0, 0)) = (2, 1, 0) + (0, 0, 0) = (2, 1, 0)$

5. Step 4
(a) $dv_4 = (0, 0, 0)$
(b) $tv_4 = (0, 0, 0)$
(c) Give $s_4 = (1, 0, 0)$
(d) $s' = (dv_{41} \times r_1) \odot (dv_{42} \times r_2) \odot (dv_{43} \times r_3) = (0 \times (0)) \odot (0 \times (0)) = (1, 1, 1)$
(e) $s_4 = (1, 0, 0) \prod (1, 1, 1) = (1, 0, 0)$
(f) $c_4 = (2, 1, 0) + (1, 0, 0), C = (2, 1, 0) + (-1, 0, 0) = (1, 1, 0)$
(g) $tv \leftarrow (0, 0, 0)$ sets $tv$ to $(0, 0, 0)$
(h) $c_4 = (1, 1, 0) + (0, 0, 0) \odot ((0, 0, 0), r) = (1, 1, 0) + (0, 0, 0) = (1, 1, 0)$
(i) $tv_4 = (0, 0, 0) \odot (1, 0, 0) \times F = (0, 0, 0) \odot (3, 0, 0) = (3, 0, 0)$
(j) $dv_4 = (0, 0, 0) + (1, 0, 0), D = (0, 0, 0) + (3, 0, 0) = (3, 0, 0)$
(k) $tv \leftarrow (3, 0, 0)$ sets $tv$ to $(0, 0, 0)$
(l) $c_4 = (1, 1, 0) + ((3, 0, 0) \odot (0, 0, 0)) = (1, 1, 0) + ((3, 0, 0) \odot (0, 0, 0)) = (1, 1, 0) + (0, 0, 0) = (1, 1, 0)$

6. Step 5
(a) $dv_5 = (2, 0, 0)$
(b) $tv_5 = (2, 0, 0)$
(c) Give $s_5 = (1, 0, 0)$
(d) $s' = (dv_{5_1} \bowtie r_1) \odot (dv_{5_2} \bowtie r_2) \odot (dv_{5_3} \bowtie r_3) = (2 \bowtie (0)) \odot (0 \bowtie (0)) = (0,1,1)$

(e) $s_5 = (1,0,0) \prod (0,1,1) = (0,0,0)$

(f) $c_5 = (1,1,0) + (0,0,0).C = (1,1,0) + (0,0,0) = (1,1,0)$

(g) $tv \leftarrow (2,0,0)$ sets $tv$ to $(0,0,0)$

(h) $c_5 = (1,1,0) + (2,0,0) \odot ((0,0,0).T)) = (1,1,0) + (0,0,0) = (1,1,0)$

(i) $tv_5 = (2,0,0) \odot (0,0,0) \times F = (2,0,0) \odot (0,0,0) = (2,0,0)$

(j) $dv_5 = (2,0,0) + (0,0,0)\cdot D = (2,0,0) + (0,0,0) = (2,0,0)$

(k) $tv' \leftarrow (2,0,0)$ sets $tv'$ to $(0,0,0)$

(l) $c_5 = (1,1,0) + ((2,0,0) \odot ((0,0,0).T)) = (1,1,0) + ((2,0,0) \odot (0,0,0)) = (1,1,0) + (0,0,0) = (1,1,0)$

7. Step 6

(a) $dv_6 = (1,0,0)$

(b) $tv_6 = (1,0,0)$

(c) Give $s_6 = (1,0,0)$

(d) $s = (dv_{6_1} \bowtie r_1) \odot (dv_{6_2} \bowtie r_2) \odot (dv_{6_3} \bowtie r_3) = (1 \bowtie (0)) \odot (0 \bowtie (0)) = (0,1,1)$

(e) $s_6 = (1,0,0) \prod (0,1,1) = (0,0,0)$

(f) $c_6 = (1,1,0) + (0,0,0).C = (1,1,0) + (0,0,0) = (1,1,0)$

(g) $tv \leftarrow (1,0,0)$ sets $tv$ to $(1,0,0)$

(h) $c_6 = (1,1,0) + (1,0,0) \odot ((1,0,0).T)) = (1,1,0) + (0,1,0) = (1,2,0)$

(i) $tv_6 = (1,0,0) \odot (0,0,0) \times F = (1,0,0) \odot (0,0,0) = (0,0,0)$

(j) $dv_6 = (1,0,0) + (0,0,0)\cdot D = (1,0,0) + (0,0,0) = (1,0,0)$

(k) $tv' \leftarrow (0,0,0)$ sets $tv'$ to $(0,0,0)$

(l) $c_6 = (1,2,0) + ((1,0,0) \odot ((0,0,0).T)) = (1,2,0) + ((1,0,0) \odot (0,0,0)) = (1,2,0) + (0,0,0) = (1,2,0)$

8. Step 7

(a) $dv_7 = (0,0,0)$

(b) $tv_7 = (0,0,0)$

(c) Give $s_7 = (1,1,0)$

(d) $s = (dv_{7_1} \bowtie r_1) \odot (dv_{7_2} \bowtie r_2) \odot (dv_{7_3} \bowtie r_3) = (0 \bowtie (0)) \odot (0 \bowtie (0)) = (1,1,1)$

(e) $s_7 = (1,1,0) \prod (1,1,1) = (1,1,0)$

(f) $c_7 = (1,2,0) + (1,1,0).C = (1,2,0) + (-1,-2,0) = (0,0,0)$

(g) $tv \leftarrow (0,0,0)$ sets $tv$ to $(0,0,0)$

(h) $c_7 = (0,0,0) + (0,0,0) \odot ((0,0,0).T)) = (0,0,0) + (0,0,0) = (0,0,0)$

(i) $tv_7 = (0,0,0) \odot (1,1,0) \times F = (0,0,0) \odot (3,2,0) = (3,2,0)$

(j) $dv_7 = (0,0,0) + (1,1,0)\cdot D = (0,0,0) + (3,2,0) = (3,2,0)$

(k) $tv' \leftarrow (3,2,0)$ sets $tv'$ to $(0,0,0)$

(l) $c_7 = (0,0,0) + ((3,2,0) \odot ((0,0,0).T)) = (0,0,0) + ((3,2,0) \odot (0,0,0)) = (0,0,0) + (0,0,0) = (0,0,0)$

9. Step 8

(a) $dv_8 = (2,1,0)$

(b) $tv_8 = (2,1,0)$

(c) Give $s_8 = (0,0,0)$
(d) $s' = (dv_{81} \triangleleft r_1) \odot (dv_{82} \triangleleft r_2) \odot (dv_{83} \triangleleft r_3) = (2 \trianglerighteq (0)) \odot (1 \trianglerighteq (0)) \odot (0 \trianglerighteq (0)) = (0, 0, 1)$

(e) $s_8 = (0, 0, 0) \prod (0, 0, 1) = (0, 0, 0)$

(f) $c_8 = (0, 0, 0) + (0, 0, 0). C = (0, 0, 0) + (0, 0, 0) = (0, 0, 0)$

(g) $tv \leftarrow (2, 1, 0)$ sets $tv$ to $(0, 1, 0)$

(h) $c_8 = (0, 0, 0) + (2, 1, 0) \odot ((2, 1, 0). T) = (0, 0, 0) + (0, 0, 1) = (0, 0, 1)$

(i) $tv_9 = (2, 1, 0) \oplus (0, 0, 0) \times F = (2, 1, 0) \oplus (0, 0, 0) = (2, 0, 0)$

(j) $dv_8 = (2, 1, 0) + (0, 0, 0), D = (2, 1, 0) + (0, 0, 0) = (2, 1, 0)$

(k) $tv' \leftarrow (2, 0, 0)$ sets $tv'$ to $(0, 0, 0)$

(l) $c_8 = (0, 0, 1) + ((2, 1, 0) \odot ((0, 0, 0). T)) = (0, 0, 1) + ((2, 1, 0) \odot (0, 0, 0)) = ((0, 0, 1) + (0, 0, 0) = (0, 0, 1)$

10. Step 9

(a) $dv_9 = (1, 0, 0)$

(b) $tv_9 = (1, 0, 0)$

(c) Give $s_9 = (0, 0, 1)$

(d) $s' = (dv_{91} \triangleleft r_1) \odot (dv_{92} \triangleleft r_2) \odot (dv_{93} \triangleleft r_3) = (1 \trianglerighteq (0)) \odot (0 \trianglerighteq (0)) = (0, 1, 1)$

(e) $s_9 = (0, 0, 1) \prod (0, 1, 1) = (0, 0, 1)$

(f) $s_9 = (0, 0, 1) + (0, 0, 1). C = (0, 0, 1) + (0, 0, 1) = (0, 0, 0)$

(g) $tv \leftarrow (1, 0, 0)$ sets $tv$ to $(1, 0, 0)$

(h) $c_9 = (0, 0, 0) + (1, 0, 0) \odot ((1, 0, 0). T) = (0, 0, 0) + (0, 1, 0) = (0, 1, 0)$

(i) $tv_9 = (1, 0, 0) \oplus (0, 0, 1) \times F = (1, 0, 0) \oplus (0, 0, 0) = (0, 0, 0)$

(j) $dv_9 = (1, 0, 0) + (0, 0, 1). D = (1, 0, 0) + (0, 0, 0) = (1, 0, 0)$

(k) $tv' \leftarrow (0, 0, 0)$ sets $tv'$ to $(0, 0, 0)$

(l) $c_9 = (0, 1, 0) + ((1, 0, 0) \odot ((0, 0, 0). T)) = (0, 1, 0) + ((1, 0, 0) \odot (0, 0, 0)) = ((0, 1, 0) + (0, 0, 0) = (0, 1, 0)$

No rules can be applied. So the computation halts. The system sends spikes during the first step and the ninth step. So the result of the computation is $9 - 1 = 8$. 

Matrix Representation of Spiking Neural P Systems with Delay
Asynchronous Extended Spiking Neural P Systems with Astrocytes

Linqiang Pan\textsuperscript{1}, Jun Wang\textsuperscript{1,2,*}, Hendrik Jan Hoogeboom\textsuperscript{2}

\textsuperscript{1} Image Processing and Intelligent Control Key Laboratory of Education Ministry
Department of Control Science and Engineering
Huazhong University of Science and Technology
Wuhan 430074, Hubei, China
lqpan@mail.hust.edu.cn, junwangjf@gmail.com

\textsuperscript{2} Leiden Institute of Advanced Computer Science, Universiteit Leiden,
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
hoogeboom@liacs.nl

Abstract. A variant of spiking neural P systems was recently investigated by the authors [Pan, Wang, Hoogeboom: Spiking Neural P Systems with Astrocytes, submitted], where astrocytes have excitatory and inhibitory influence on synapses. In this work, we consider this new system in the non-synchronized (i.e., asynchronous) mode: in any step, when a neuron is enabled, it is not obligatorily fired, making a global clock dispensable. It is proved that asynchronous spiking neural P systems with astrocytes are universal (when using extended rules).

Keywords: Spiking Neural P System, Asynchronous Mode, Astrocyte, Turing Computability.

1 Introduction

Spiking neural P systems (SN P systems, for short) are a class of distributed and parallel computation models inspired by the way neurons communicate by means of electrical impulses of identical shape (called spikes). SN P systems were introduced in [5], and then investigated in a large number of papers. Readers can refer to the recent handbook [4] for general information in this area.

Briefly, an SN P system consists of a set of neurons placed as the nodes of a directed graph. The content of each neuron consists of a number of copies of a single object type, called the spike. The rules assigned to neurons allow a neuron to send information to all other neurons that are connected to it.

Classically, an SN P system works in a synchronized manner. A global clock is assumed, and in each time unit, for each neuron with applicable rules one of these rules is nondeterministically chosen, which are then applied on the tick of the clock, for all neurons at the same time. While globally parallel, the work of the system is sequential in each neuron as (at most) one rule is applied in

\* Corresponding author
each neuron. Output can be defined in the form of the spike train produced by a specified output neuron, then measuring the timing between events, or counting the number of spikes in that neuron.

However, both from a mathematical point of view and from a neuro-biological point of view, it is rather natural to consider non-synchronized systems, where the application of rules is not obligatory, and no global clock is assumed. The neuron may remain unfired, maybe receiving spikes from the neighbouring neurons. Thus the unused rule may be used later, or it may loose its applicability. If further spikes made the rule non-applicable, then the computation continues in the new circumstances (maybe other rules are enabled now). With this motivation, asynchronous SN P systems were introduced in [2], and it was proved that asynchronous SN P systems with extended rules are equivalent with Turing machines.

Recently, a variant of spiking neural P systems with astrocytes was considered in [7], where an astrocyte can sense at the same time the spike traffic along several neighboring synapses. In an SN P system with astrocytes, each astrocyte \( \text{ast} \) has a given threshold \( t \). Suppose that there are \( k \) spikes passing along the neighboring synapses. If \( k \) is larger than the threshold \( t \), then the astrocyte \( \text{ast} \) has an inhibitory influence on the neighboring synapses, and the \( k \) spikes are suppressed (that is, they are removed from the system). If \( k \) is less than the threshold \( t \), then the astrocyte \( \text{ast} \) has an excitatory influence on the neighboring synapses, the \( k \) spikes survive and pass to the destination neurons. If \( k \) equals \( t \), then the astrocyte \( \text{ast} \) non-deterministically chooses an inhibitory or excitatory influence on the neighboring synapses.

Actually, astrocytes were already introduced into spiking neural P systems in [1, 9]. The functioning of astrocytes defined in [1, 7, 9] are different, although all of them are inspired by living astrocytes. In [1], two kinds of astrocytes are defined: excitatory astrocytes and inhibitory astrocytes. Astrocytes defined in [9] are a particular case of those considered in [1], with an inhibitory role; furthermore, the use of astrocytes defined in [9] adds a new degree of non-determinism to the functioning of the system, by the branching due to the non-deterministic choice of the surviving spike.

In this work, building [2], on we prove that spiking neural P systems with astrocytes are equivalent with Turing machines in the non-synchronized case. In our proof of universality, forgetting rules are not used, and they are actually replaced by the inhibitory influence of astrocytes. Moreover, all neurons in our construction work in a deterministic way (the nondeterminism in choosing the rules of a neuron is compensated by the nondeterminism from astrocytes).

In the universality proof in [2], the SUB modules of systems contain a parameter \( T \), where \( T \) depends on the maximum number of SUB instructions that act on a same counter in the simulated register machine. However, in the present work, all modules of systems are constructed in a uniform way in the sense that all modules are independent of the particular register machine that is simulated (more specifically, for different register machines the combination of modules used is different; while the modules used are same).
The rest of this paper is organized as follows. In the next section, we introduce some necessary prerequisites. In Section 3, we present spiking neural P systems with astrocytes. In Section 4, we prove that spiking neural P systems with astrocytes are universal in non-synchronized case. Final remarks are presented in Section 5.

2 Prerequisites

Readers can refer to [10] for basic language and automata theory, as well as to [8] for basic membrane computing. We here only fix some necessary notions and notations.

For an alphabet $V$, $V^*$ denotes the set of all finite strings over $V$, with the empty string denoted by $\lambda$. The set of all nonempty strings over $V$ is denoted by $V^+$. When $V = \{a\}$ is a singleton, then we write simply $a^*$ and $a^+$ instead of $\{a\}^*$, $\{a\}^+$.

Regular expressions are built starting from $\lambda$ and single symbols using the operators union ($\cup$), concatenation ($\cdot$) and star ($^*$). The language represented by expression $E$ is denoted by $L(E)$, where $L(\lambda) = \emptyset$.

By $NRE$ we denote the families of Turing computable sets of numbers. (Thus, $NRE$ is the family of length sets of recursively enumerable languages.)

A register machine is a construct $M = (m, H, l_0, l_h, I)$, where $m$ is the number of registers (each holding a natural number), $H$ is the set of instruction labels, $l_0$ is the start label (labeling an ADD instruction), $l_h$ is the halt label (assigned to instruction HALT), and $I$ is the set of instructions. Each label from $H$ labels exactly one instruction from $I$, thus precisely identifying it. The instructions are of the following form (and have the given intended meaning):

- $l_i : (\text{ADD}(r), l_j, l_k)$ (add 1 to register $r$ and then nondeterministically go to one of the instructions with labels $l_j, l_k$),
- $l_i : (\text{SUB}(r), l_j, l_k)$ (if register $r$ is non-zero, then subtract 1 from it, and go to the instruction with label $l_j$; otherwise, go to the instruction with label $l_k$),
- $l_h : \text{HALT}$ (the halt instruction, the computation stops).

A register machine $M$ computes (generates) a number $n$ in the following way. The register machine starts with all registers empty (i.e., storing the number zero). It applies the instruction with label $l_0$ and proceeds to apply instructions as indicated by labels (and, in the case of SUB instructions, by the contents of registers). If the register machine reaches the halt instruction, then the number $n$ stored at that time in the first register is said to be computed by $M$. The set of all numbers computed by $M$ is denoted by $N(M)$. It is known that register machines compute all sets of numbers which are Turing computable, hence they characterize $NRE$ [6].

Without loss of generality, it can be assumed that $l_0$ labels an ADD instruction and that in the halting configuration all registers different from the first one are empty, and that the output register is never decremented during the computation (its content is only added to).
We use the following convention. When the power of two number generating/accepting devices $D_1$ and $D_2$ are compared, number zero is ignored; that is, we write $N(D_1) = N(D_2)$ when we actually mean that $N(D_1) - \{0\} = N(D_2) - \{0\}$ (this corresponds to the usual practice of ignoring the empty string in language and automata theory).

3 Spiking Neural P Systems with Astrocytes

In this section, we introduce the variant of asynchronous spiking neural P systems with astrocytes (ASNPA systems, for short). For more details on such kind of systems, please refer to [2] and [7].

An (extended) spiking neural P system with astrocytes, of degree $m \geq 1$, $\ell \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, \text{syn}, \text{ast}_1, \ldots, \text{ast}_\ell, \text{out}),$$

where:

- $O = \{a\}$ is the singleton alphabet ($a$ is called spike);
- $\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, where:
  - a) $n_i \geq 0$ is the initial number of spikes contained in $\sigma_i$;
  - b) $R_i$ is a finite set of extended rules of the following form:
    $$E/a^c \rightarrow a^p$$
    where $E$ is a regular expression over $a$, and $c \geq 1$, $p \geq 1$ with $c \geq p$.
- $\text{syn} \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin \text{syn}$ for $1 \leq i \leq m$ (synapses between neurons);
- $\text{ast}_1, \ldots, \text{ast}_\ell$ are astrocytes, of the form $\text{ast}_i = (\text{syn}_{\text{ast}_i}, t_i)$, where $1 \leq i \leq \ell$, $\text{syn}_{\text{ast}_i} \subseteq \text{syn}$ is the set of synapses controlled by the astrocyte $\text{ast}_i$, $t_i \in \mathbb{N}$ is the threshold of the astrocyte $\text{ast}_i$;
- $\text{out} \in \{1, 2, \ldots, m\}$ indicate the output neuron.

The rules $E/a^c \rightarrow a^p$ with $p \geq 1$ are called (extended) firing (we also say spiking) rules, and they are applied as follows. If the neuron $\sigma_i$ contains $k$ spikes, and $a^k \in L(E)$, $k \geq c$, then rule $E/a^c \rightarrow a^p \in R_i$ can be applied. This means consuming (removing) $c$ spikes (leaving $k-c$ spikes in neuron $\sigma_i$), the neuron is fired, sending $p$ spikes out along all outgoing synapses. These spikes then reach the neighbouring neurons, unless they are intercepted by one of the astrocytes, as explained shortly below. If $L(E) = \{a^c\}$, then the rule is written in the simplified form $a^c \rightarrow a^p$.

In the non-synchronized case, considered here, the definition of a computational step in an ASNPA system is easy. In each moment, any neuron is free to use an applicable rule or not. Hence, if a rule in neuron $\sigma_i$ is enabled at step $t$,

---

1 We adhere to the practice where the number of spikes sent is not larger than the number of spikes removed from the neuron. This turns out to be only a technical constraint: a set of ‘parallel’ neurons is used to increase the number of spikes where necessary.
it is not necessarily applied at that step, the neuron can remain still in spite of the fact that it contains rules which are enabled by its contents. If the enabled rule in neuron $\sigma_i$ is not applied, and neuron $\sigma_i$ receives new spikes, making the rule non-applicable, then the computation continues in the new circumstances (where maybe other rules are enabled).

Note that in “classical” membrane computing a global clock is assumed, marking the time for the whole system, and the functioning of the system is synchronized. In the synchronized mode, in each time unit, if a neuron $\sigma_i$ is enabled, then one of its rules must be used. It is possible that two or more rules in a neuron are applicable at the same time, and in that case, only one of them is chosen non-deterministically. Thus the neurons together work in parallel (synchronously), while each separate neuron sequentially processes its spikes, using only one rule in each time unit.

Astrocytes work as follows. An astrocyte can sense the spike traffic along the neighboring synapses. For an astrocyte $ast_i$, suppose that there are $k$ spikes passing along the neighboring synapses in $syn_{ast_i}$. If $k > t_i$, then the astrocyte $ast_i$ has an inhibitory influence on the neighbouring synapses, and the $k$ spikes are suppressed (that is, the spikes are removed from the system). If $k < t_i$, then the astrocyte $ast_i$ has an excitatory influence on the neighboring synapses, all spikes survive and pass to the destination neurons. If $k = t_i$, then the astrocyte $ast_i$ non-deterministically chooses an inhibitory or excitatory influence on the neighboring synapses.

It is important to point out that when a neuron spikes, its spikes immediately leave the neuron along all synapses. This in particular means that (i) if the spikes are not suppressed by any astrocyte, these spikes reach the target neurons simultaneously (as in the synchronized system, no time is needed for passing along a synapse from one neuron to another neuron); (ii) if any astrocyte has inhibitory influence on the synapse, then these spikes are suppressed simultaneously, which means that astrocytes remove the spikes in the synchronized way.

It is possible that two or more astrocytes control the same synapse. In this case, if all these astrocytes have excitatory influence on the synapse, then the spikes along this synapse can survive and pass to the destination neurons; if one of these astrocytes has inhibitory influence on the synapse, then the spikes along this synapse are suppressed and removed from the system.

A configuration of the system is described by the number of spikes present in each neuron. The initial configuration is defined by the number of initial spikes $n_1, \ldots, n_m$. Using the rules as described above, one can define transitions among configurations. Any sequence of transitions starting from the initial configuration is called a computation. A computation halts when it reaches a configuration where no rule can be used.

Because in asynchronous SN P systems with astrocytes, an enabled rule can be applied at any moment, the result of a computation can no longer be reliably defined in terms of the steps between two consecutive spikes as in one of the standard SN P system definitions. Therefore, in this work, the result of a computation is defined as the total number of spikes sent into the environment.
by the output neuron. Specifically, if there is a halting computation of an SNP system where the output neuron sends out exactly \( n \) spikes, then the system generates the number \( n \).

Halting computations that send no spike out can be considered as generating number zero, but, in this work, we adopt the convention to ignore number zero when the computation power of two devices is compared. The reason is technical: we want to consider blocked computations without output as failure.

Note that the definitions in this paper specify systems with extended rules (introduced by [3], meaning that more than one spike can be emitted) but without forgetting rules (meaning that at least one spike must be emitted when firing). Astrocytes are an effective alternative method for removing spikes, so forgetting rules are not necessary in our considerations.

We denote by \( N(\Pi) \) the set of numbers generated in the asynchronous way by an ASNPA system \( \Pi \). By \( N_{gen}^{asyn}SNPA(extend) \) we denote the family of such sets of numbers generated by ASNPA systems, using extended rules.

4 Universality of ASNPA Systems

In this section, ASNPA systems are proved to be universal in the generative mode.

**Theorem 1.** \( NRE = N_{gen}^{asyn}SNPA(extend) \).

**Proof.** We show that \( NRE \subseteq N_{gen}^{asyn}SNPA(extend) \); the converse inclusion is straightforward (simulating the system by a Turing machine). To this aim, we use the characterization of \( NRE \) by means of register machines used in the generative mode. Let us consider a register machine \( M = (m, H, l_0, l_h, I) \). As introduced in Section 2, without any loss of generality, we may assume that in the halting configuration, all registers different from the output register 1 are empty, and that this register is never decremented during a computation. In what follows, a specific ASNPA system \( \Pi \) will be constructed to simulate the register machine \( M \).

Each register \( r \) of \( M \) will be implemented by a neuron \( \sigma_r \) in \( \Pi \), and if the register contains the number \( n \), then the associated neuron will have \( 6n + 6 \) spikes. A neuron \( \sigma_{l_i} \) is associated with each label \( l_i \in H \). We construct modules ADD and SUB to simulate the instructions of \( M \), connecting the appropriate label-neurons, and introducing further auxiliary neurons \( \sigma_{l(j)}, j = 1, 2, 3, \ldots \).

Finally an output module FIN is constructed to output computation results.

The modules will be specified in a graphical form. Neurons are denoted by rounded rectangles with the number of initial spikes inside; arrows between these rounded rectangles represent the synapses. An astrocyte is denoted by a rhombic box with "arms" touching the synapses; each arm indicates that the astrocyte controls the spike traffic of the corresponding touched synapse; the equation \( t = k \) inside the rhombic box denotes that the astrocyte has threshold \( k \).

In the initial configuration, all neurons are empty except that neuron \( \sigma_{l_0} \) associated with label \( l_0 \) of \( M \) has two spikes inside to set the system working.
and each counter neuron $\sigma_r$ contains 6 spikes representing empty counters. In general, when a label-neuron $\sigma_{l_i}, l_i \in H$, has two spikes inside, then it becomes active and the module associated with instruction $l_i$ starts to work, simulating the instruction. Simulation is transferred to another instruction when the next label-neuron obtains two spikes.

**Module ADD** – simulating an ADD instruction $l_i : (\text{ADD}(r), l_j, l_k)$.

Module ADD, shown in Figure 1, is composed of 2 astrocytes and 12 neurons: neuron $\sigma_r$ for register $r$; neurons $\sigma_{l_1}, \sigma_{l_2}, \sigma_{l_k}$ for labels $l_1, l_j, l_k$; and 8 auxiliary neurons $\sigma_{l_1^{(1)}}, \ldots, \sigma_{l_1^{(6)}}$. The astrocyte $\text{ast}_{l_1^{(1)}}$ touches two synapses $(\sigma_{l_1^{(5)}}, \sigma_{l_1^{(7)}})$ and $(\sigma_{l_1^{(5)}}, \sigma_{l_1^{(8)}})$, and its threshold is 6. The astrocyte $\text{ast}_{l_1^{(2)}}$ touches synapse $(\sigma_{l_1^{(8)}}, \sigma_{l_k})$ and its threshold is 3.

Let us assume that at a computational step instruction $l_i : (\text{ADD}(r), l_j, l_k)$ has to be simulated, with two spikes present in neuron $\sigma_{l_1}$ and no spike in any other neurons, except in those neurons associated with registers.

Having two spikes inside, neuron $\sigma_{l_1}$ may fire, sending 2 spikes out. These spikes will simultaneously go to neurons $\sigma_{l_1^{(1)}}, \sigma_{l_1^{(2)}}$ and $\sigma_{l_1^{(3)}}$. Each of these three neurons can fire, and indeed must fire before neuron $\sigma_{l_1^{(4)}}$ is enabled. Then, with 6 spikes inside, neuron $\sigma_{l_1^{(4)}}$ sends 6 spikes to neurons $\sigma_r$ (simulating the increase of the value of register $r$ with 1) and $\sigma_{l_1^{(5)}}$. With 6 spikes inside, neuron $\sigma_{l_1^{(5)}}$ can use its rule $a^6 \rightarrow a^3$, sending 3 spikes to each of the neurons $\sigma_{l_1^{(6)}}, \sigma_{l_1^{(7)}}$ and $\sigma_{l_1^{(8)}}$.

At that moment there are 6 spikes passing along the synapses controlled by astrocyte $\text{ast}_{l_1^{(1)}}$ which has threshold 6. The astrocyte $\text{ast}_{l_1^{(1)}}$ non-deterministically chooses an inhibitory or excitatory influence on its controlled synapses. This choice determines whether the next label-neuron activated will eventually be $\sigma_{l_j}$ or $\sigma_{l_k}$.

If astrocyte $\text{ast}_{l_1^{(1)}}$ chooses an inhibitory influence on its controlled synapses, then only neuron $\sigma_{l_1^{(6)}}$ receives 3 spikes from neuron $\sigma_{l_1^{(5)}}$. In turn it will send 2 spikes to neuron $\sigma_{l_1^{(8)}}$, which then can send its 2 spikes to neuron $\sigma_{l_1}$ at one moment. The synapse $(\sigma_{l_1^{(6)}}, \sigma_{l_k})$ is controlled by astrocyte $\text{ast}_{l_1^{(2)}}$ but the number of spikes is less than the threshold 3 and the 2 spikes reach neuron $\sigma_{l_k}$. At that moment system $H$ starts to simulate the instruction $l_k$ of $M$.

If astrocyte $\text{ast}_{l_1^{(1)}}$ chooses an excitatory influence on the synapses it controls, then 3 spikes reach each of the neurons $\sigma_{l_1^{(6)}}, \sigma_{l_1^{(7)}}$ and $\sigma_{l_1^{(8)}}$. With 3 spikes inside, neuron $\sigma_{l_1^{(7)}}$ can send 2 spikes to neuron $\sigma_{l_j}$ starting the simulation of the instruction with label $l_j$ of $M$. With 3 spikes inside, neuron $\sigma_{l_1^{(8)}}$ cannot apply any rule until further 2 spikes coming from neuron $\sigma_{l_1^{(6)}}$ arrive in neuron $\sigma_{l_1^{(8)}}$. Then, having 5 spikes inside, neuron $\sigma_{l_1^{(8)}}$ can send 5 spikes to neuron $\sigma_{l_k}$ at one moment. However, these 5 spikes passing along the synapse $(\sigma_{l_1^{(8)}}, \sigma_{l_k})$ are suppressed by astrocyte $\text{ast}_{l_1^{(2)}}$ and vanish from the system.
Fig. 1. Module ADD simulating \( l_1 : (\text{ADD}(r), l_j, l_k) \)
Note however that the system is asynchronous. There is no way to force the module to fire any of the neurons $\sigma_{i(6)}$ and $\sigma_{i(8)}$. Thus the possibility exists that 3 new spikes arrive (the next time that instruction $l'_i$ is simulated) to neuron $\sigma_{i(6)}$ and/or neuron $\sigma_{i(8)}$ while that neuron is not yet empty. A careful case analysis now shows that this leads to more than 3 or 5 spikes in neuron $\sigma_{i(6)}$ or neuron $\sigma_{i(8)}$, respectively. This effectively means that label $l_k$ becomes unreachable, blocking computations taking that branch. Most importantly however, it does not introduce halting but unwanted computations.

Therefore, from firing neuron $\sigma_{l_i}$, the system may non-deterministically fire one of neurons $\sigma_{l_i}$ and $\sigma_{l_k}$, which correctly simulates the ADD instruction $l_i : (\text{ADD}(r), l_j, l_k)$. Any other computation will stop without generating an output.

**Module SUB** — simulating a SUB instruction $l_i : (\text{SUB}(r), l_j, l_k)$.

Module SUB, shown in Figure 2, is composed of 3 astrocytes $\text{ast}_{i(1)}^l$, $\text{ast}_{i(2)}^l$, $\text{ast}_{i(3)}^l$ and 18 neurons: neurons $\sigma_i$ for register $r$, neuron $\sigma_{l_i}$, $\sigma_{l_j}$, $\sigma_{l_k}$ for labels $l_i$, $l_j$, $l_k$, and 14 auxiliary neurons $\sigma_{i(1)}$, $\sigma_{i(2)}$, . . . , $\sigma_{i(14)}$.

An instruction $l_i$ is simulated in $\Pi$ in the following way. Initially, neuron $l_i$ has two spikes, and the other neurons are empty, except neurons associated with registers. Assume that at one moment neuron $\sigma_{l_i}$ fires and sends 2 spikes to neurons $\sigma_{i(1)}$ and $\sigma_{i(2)}$. Neuron $\sigma_{i(3)}$ will produce one spike and send it to neuron $\sigma_r$. There are the following two cases, depending whether the counter $r$ contains a positive value or not.

**Positive counter.** If neuron $\sigma_r$ has $6n + 6$ ($n > 0$) spikes (corresponding to the fact that the number stored in register $r$ is positive), then after receiving one spike from neuron $\sigma_{i(1)}$, neuron $\sigma_r$ has $6n + 7$ spikes, and the enabled rule $a^7(a^6)^+ / a^7 \rightarrow a^6$ will fire, sending 6 spikes to neurons $\sigma_{i(2)}$ and $\sigma_{i(3)}$, respectively. For neuron $\sigma_{i(2)}$, there are following two cases.

(a) Neuron $\sigma_{i(2)}$ has not consumed the first 2 spikes (received from neuron $\sigma_{l_i}$) before getting the second 6 spikes from neuron $\sigma_r$. Then, with 8 spikes inside, neuron $\sigma_{i(2)}$ will send 6 spikes to neuron $\sigma_{i(3)}$. This number is larger than the threshold 5 of astrocyte $\text{ast}_{i(1)}^l$, thus the spikes are suppressed. With 6 spikes (from neuron $\sigma_r$) inside, neuron $\sigma_{i(3)}$ sends 4 spikes to neurons $\sigma_{i(4)}$ and $\sigma_{i(5)}$, respectively. Again however, all the spikes sent out from neuron $\sigma_{i(3)}$ are suppressed by astrocyte $\text{ast}_{i(2)}^l$ as the total number of 8 spikes passing along the synapses is larger than the threshold 7 of astrocyte $\text{ast}_{i(2)}^l$. In this case, neither of the neurons $\sigma_{l_i}$ and $\sigma_{l_k}$ receive a spike, which means that computations in system $\Pi$ abort, outputting no spike into the environment.

(b) Neuron $\sigma_{i(2)}$ receives the first 2 spikes from neuron $\sigma_{l_i}$, and rule $a^2 \rightarrow a^2$ is applied before the 6 spikes from neuron $\sigma_r$ arrive in neuron $\sigma_{i(2)}$, then later these 6 spikes will be consumed and 4 spikes will be sent to neuron $\sigma_{i(3)}$. These numbers are less than the threshold and all spikes will arrive in
Fig. 2. Module SUB simulating $l_i : (\text{SUB}(r), l_j, l_k)$
neuron $\sigma_{l(3)}$. When neuron $\sigma_{l(3)}$ first receives the 2 spikes from neuron $\sigma_{l(2)}$ it cannot apply any rule. When it receives the 6 spikes from neuron $\sigma_{r}$, the contents of $\sigma_{l(3)}$ will be 8 spikes, while $\sigma_{l(2)}$ contains 6 spikes. Now both $\sigma_{l(2)}$ and $\sigma_{l(3)}$ are enabled. There are two situations:

- Enabled rule $a^8 \rightarrow a^3$ is not applied before the 4 spikes from neuron $\sigma_{l(2)}$ arrive in neuron $\sigma_{l(3)}$. Then neuron $\sigma_{l(3)}$ accumulates 12 spikes, the number of spikes is greater than 8, thus, no rule is enabled and the computation aborts.

- Neuron $\sigma_{l(3)}$ applies the rule $a^8 \rightarrow a^3$ before receiving the 4 spikes from neuron $\sigma_{l(2)}$. Then 3 spikes are sent out to both neurons $\sigma_{l(4)}$ and $\sigma_{l(5)}$. So, there are 6 spikes passing along the synapses controlled by astrocyte $ast_{l(r)}^{(2)}$. The number 6 is less than the threshold 7 of this astrocyte, hence these spikes can reach neurons $\sigma_{l(4)}$ and $\sigma_{l(5)}$. With an odd number of spikes inside (an even number plus the three just received), neuron $\sigma_{l(4)}$ can fire by using the rule $a(a^2)^+ / a^3 \rightarrow a^2$. Now neuron $\sigma_{l(r)}$ receives 2 spikes, and the system $H$ starts to simulate instruction $l_j$ of $M$. At the same time neuron $\sigma_{l(4)}$ again has an even number of spikes. In the other branch, with 3 spikes inside, neuron $\sigma_{l(5)}$ can send 3 spikes to neuron $\sigma_{l(6)}$. However, these 3 spikes are suppressed by astrocyte $ast_{l(r)}^{(3)}$ and do not reach their destination neuron $\sigma_{l(6)}$. Note that, after sending 3 spikes out, neuron $\sigma_{l(3)}$ will then receive 4 spikes from neuron $\sigma_{l(2)}$ and produce 4 spikes, but these spikes are suppressed by astrocyte $ast_{l(r)}^{(2)}$ and cannot reach neurons $\sigma_{l(4)}$ and $\sigma_{l(5)}$.

### Zero counter

If neuron $\sigma_{r}$ initially has 6 spikes (corresponding to an empty register $r$), then at one moment after receiving the additional spike from $\sigma_{l(1)}$, neuron $\sigma_{r}$ has 7 spikes inside, sending 5 spikes to $\sigma_{l(2)}$ and $\sigma_{l(3)}$, respectively. Neurons $\sigma_{l(2)}$ and $\sigma_{l(3)}$ are designed in such a way that their behaviour in the case of an empty counter is completely analogous to the non-zero case described above, except that the crucial numbers 6 and 8 of spikes are to be replaced by 5 and 7, leading to the same way incorrect computations abort, while a correct simulation will fire two spikes from $\sigma_{l(4)}$. We omit the cases where the computation aborts, and only consider the correct branch (corresponding to the second item from (b) above in the non-zero case).

(b) Consider that neuron $\sigma_{l(2)}$ contains 5 spikes received from $\sigma_{r}$, which means that it already sent 2 spikes to neuron $\sigma_{l(3)}$. With 7 spikes inside, neuron $\sigma_{l(3)}$ can apply the rule $a^7 \rightarrow a^2$.

- Neuron $\sigma_{l(3)}$ first consumes 7 spikes and sends 2 spikes to neurons $\sigma_{l(4)}$ and $\sigma_{l(5)}$, respectively, and later sends 4 spikes out by using the rule $a^4 \rightarrow a^1$, then each neuron $\sigma_{l(4)}$ and $\sigma_{l(5)}$ receives only 2 spikes from
neuron \( \sigma_l^{(3)} \) (the 4 spikes sent out from neuron \( \sigma_l^{(3)} \)) are suppressed by astrocyte \( \text{ast}_l^{(2)} \). Neuron \( \sigma_l^{(4)} \) gets 2 spikes and keeps an even number of spikes, thus, no rule can be used and neuron \( \sigma_l^i \) receive no spikes. With 2 spikes inside, neuron \( \sigma_l^{(5)} \) will send one spike to neuron \( \sigma_l^{(6)} \), which of course is not blocked by astrocyte \( \text{ast}_l^{(3)} \). Neuron \( \sigma_l^{(6)} \) will send a spike to each of the neurons \( \sigma_l^{(7)}, \ldots, \sigma_l^{(12)} \). The six neurons will send their spike to neuron \( \sigma_l^{(13)} \). With 6 spikes inside, neuron \( \sigma_l^{(13)} \) can send 6 spikes to neuron \( \sigma_r \) (reestablishing the number 0 in register \( r \)) and \( \sigma_l^{(14)} \). Neuron \( \sigma_l^{(14)} \) will send 2 spikes to neuron \( \sigma_{l_k} \). This neuron becomes active, and the system \( \Pi \) can start to simulate instruction \( l_k \) of \( M \).

The simulation of SUB instruction is correct: system \( \Pi \) starts from \( \sigma_l \) and ends in \( \sigma_{l_4} \) (if the number stored in register \( r \) is greater than 0 and decreased by one), or in \( \sigma_{l_k} \) (if the number stored in register \( r \) is 0).

Note that there is no interference between the ADD modules and the SUB modules, other than correctly firing the neurons \( \sigma_l \) or \( \sigma_{l_k} \), which may label instructions of the other kind. However, it is possible to have interference between two SUB modules. Specifically, if there are several SUB instructions \( l_i \) that act on register \( r \), then when instruction \( l_i : (\text{SUB}(r), l_j, l_k) \) is simulated, neurons \( \sigma_l^{(2)} \) and \( \sigma_l^{(3)} \) not involved in the active instruction that is simulated nevertheless receive 6 or 5 spikes from neuron \( \sigma_r \). We show that this does not lead to unwanted halting computations. In all cases, if the simulation does not continue as desired, a module will be blocked, which then forces the computation to abort when that module later might be started. Consider the number of spikes contained in each neurons \( \sigma_l^{(2)} \) and \( \sigma_l^{(3)} \) before receiving these spikes from neurons \( \sigma_r \), we distinguish the following five cases.

1. If both neurons \( \sigma_l^{(2)} \) and \( \sigma_l^{(3)} \) are empty before receiving 6 or 5 spikes from neuron \( \sigma_r \), then after receiving these spikes, both of them contain 6 or 5 spikes (as we will see below, this is the initial state of case (2)). With 6 or 5 spikes inside, neuron \( \sigma_l^{(2)} \) sends 4 spikes to neuron \( \sigma_l^{(3)} \) and these spikes can reach their destination neuron \( \sigma_l^{(13)} \). For neuron \( \sigma_l^{(3)} \), there are two possible situations:
   a. If neuron \( \sigma_l^{(3)} \) consumes the first received 6 or 5 spikes from neuron \( \sigma_r \) before the 4 spikes from neuron \( \sigma_l^{(2)} \) arrive, then it will send 4 spikes out and contains 4 spikes (this is the initial state of case (3)). With 4 spikes inside, neuron \( \sigma_l^{(3)} \) can send 4 spikes out by using the rule \( a^4 \rightarrow a^4 \). However, all the spikes sent out from neuron \( \sigma_l^{(3)} \) are suppressed by astrocyte \( \text{ast}_l^{(2)} \) and none can reach their destination neurons \( \sigma_l^{(4)} \) and \( \sigma_l^{(5)} \).
   b. If neuron \( \sigma_l^{(3)} \) does not consume the first received 6 or 5 spikes before the 4 spikes arrive, then with 10 or 9 spikes inside (this is the initial state of case (4)), neuron \( \sigma_l^{(3)} \) cannot apply any rule.
(2) If both neurons $\sigma_{l(2)}$ and $\sigma_{l(3)}$ have 6 or 5 spikes inside before receiving 6 or 5 spikes from neuron $\sigma_r$, then after receiving these spikes, both these neurons contain 12, 11 or 10 spikes, and no rule can be used.

(3) If neuron $\sigma_{l(3)}$ has 0 spikes and neuron $\sigma_{l(2)}$ has 6 or 5 spikes inside before receiving 6 or 5 spikes from neuron $\sigma_r$, then neuron $\sigma_{l(3)}$ can send 4 spikes to neuron $\sigma_{l(2)}$. With 10 or 9 spikes inside, neuron $\sigma_{l(3)}$ can send any rule even it receives 4 spikes from neuron $\sigma_{l(2)}$.

(4) If neuron $\sigma_{l(3)}$ has 0 spikes and neuron $\sigma_{l(2)}$ has 10 or 9 spikes inside before receiving 6 or 5 spikes from neuron $\sigma_r$, then similar to case (3), neuron $\sigma_{l(2)}$ sends 4 spikes to neuron $\sigma_{l(3)}$ and neuron $\sigma_{l(3)}$ cannot apply any rule.

(5) If neuron $\sigma_{l(3)}$ has 6 or 5 spikes and neuron $\sigma_{l(2)}$ has 0 spikes inside (that is, after simulating the SUB instruction $l_1 : (\text{SUB}(r), l_1, l_k)$, it is possible that there are 6 or 5 spikes left in neuron $\sigma_{l(2)}$) before receiving 6 or 5 spikes from neuron $\sigma_r$, then after receiving these spikes, neuron $\sigma_{l(2)}$ contains 12, 11 or 10 spikes and cannot apply any rule. Neuron $\sigma_{l(3)}$ can send 4 spikes out but all the spikes are suppressed by astrocyte $\text{ast}_{l(2)}$.

Therefore, the only computations in $\Pi$ which can reach the neuron $\sigma_{l(h)}$ associated with the halting instruction of $M$ are the computations which correctly simulate the instructions of $M$ and correspond to halting computations in $M$.

**Module FIN** – outputting the result of a computation.

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\text{a}^2 \rightarrow \text{a}^2
\end{array}
\end{array}
\end{array} & \text{ } & \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\text{a}^2 \rightarrow \text{a}
\end{array}
\end{array}
\end{array} & \text{ } & \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\text{a}^7(\text{a}^5)^1/\text{a}^6 \rightarrow \text{a}
\end{array}
\end{array}
\end{array}
\end{array}
\end{align*}
\]

**Fig. 3.** Module FIN (outputting the result of the computation)

Module FIN is shown in Figure 3. Assume that the computation in $M$ halts, which means that the halting instruction is reached, and the number stored in register 1 is $n$ (we may assume that $n \geq 1$, because number 0 is ignored when the computation power of computing devices is compared). This means that neuron $\sigma_{l_h}$ in $\Pi$ has two spikes inside and rule $a^2 \rightarrow a^2$ is enabled; neuron $\sigma_1$ has $6n+6$ spikes. When neuron $\sigma_{l_h}$ fires it sends 2 spikes to neuron $\sigma_{l(3)}$ which then (eventually) sends one spike to neuron $\sigma_1$, corresponding to the first register of $M$. From now on, neuron $\sigma_1$ can fire, sending out one spike for each 6 spikes present in it. If the number of spikes in neuron $\sigma_1$ is 7, then no rule is enabled in neuron $\sigma_1$. The computation in $\Pi$ ends, and the number of spikes sent into the environment by system $\Pi$ is $n$, which is exactly the number stored in register 1 of $M$ when the computation of $M$ halts.
From the description of the modules and their work, it is clear that the register machine \( M \) is correctly simulated by system \( \Pi \). Therefore, \( N(\Pi) = N(M) \). This completes the proof.

5 Conclusions and Remarks

In this work, we have considered spiking neural P systems with astrocytes used in a non-synchronized way: if a rule is enabled at some step, this rule is not obligatorily immediately used. Further spikes may make the rule non-applicable. Such systems are comparable to Petri nets which are also used in an asynchronous way. Without further features Petri nets are not computationally complete. One such feature is the use of inhibitor arcs in the net which indicate that a transition may fire only if the associated place is empty. We have proved that asynchronous SN P systems with astrocytes are universal. This can be compared to the case of Petri nets by observing that the astrocyte has an inhibiting kind of action.

There remain several open problems about asynchronous SN P systems with astrocytes. We just list two of them. We have used extended rules and do not know whether or not asynchronous SN P systems with astrocytes are Turing complete when using only standard rules. Second, the nondeterminism in our systems originates both from the uncertainty of the moment of firing and the two possible actions when the number of spikes equals the threshold of the astrocyte (following our earlier paper [7]). It would be natural to concentrate the nondeterminism in a single feature, i.e., to make the astrocytes purely deterministic.

Acknowledgements

We thank the three referees at CMC12 for their careful reading of the paper, and in particular for bringing relevant references to our attention.

The work of L. Pan and J. Wang was supported by National Natural Science Foundation of China (61033003 and 30870826), Ph.D. Programs Foundation of Ministry of Education of China (20100142110072), Fundamental Research Funds for the Central Universities (2010ZD001), and Natural Science Foundation of Hubei Province (2008CDB113 and 2008CDB180).

References

Simulation of P systems with string objects

Veaceslav Macari⋆, Galina Magariu, and Tatiana Verlan

Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Academiei, 5, Chişinău, MD-2028 Moldova
vmacari@gmail.com, {gmagariu, tverlan}@math.md

Abstract. In this paper we present a simulator (beta version) for nondeterministic P systems with string replication. The details of representation of P system description in simulator interface and available transitions, visualization of P system configuration and computation tree are discussed.

Keywords: nondeterministic P system, string replication, simulator, P system configuration, computation tree, evolving process

1 Introduction

During last years, there were obtained a lot of significant and interesting theoretical results in membrane computing – the young branch of formal computation [1-4]. Along with this process there were developed simulators for testing, demonstrating, and better understanding of these results and for their application for some problems solution as well [5,6]. Due to the domain active progress, new types of P systems appearing and information technologies development the attempts to construct new simulators for P systems are undertaken.

Since P systems working with string objects are not sufficiently covered by already existing simulators, the authors have developed the simulator for P systems with string replication [7] in the framework of P-Lingua environment [8]. The authors experience [7] has shown that when the P systems with string objects (especially the nondeterministic ones) are simulated, significant hardware computational resources at run time are required. Therefore the framework for simulator development has to have convenient means of these computational resources distribution at the stage of simulation. So the authors decided to use .NET framework with C# and available libraries for parallel processing which will allow coping with these problems.

The version of the simulator, presented in the paper, is designed for P systems with string replication and intended to try-out the internal P systems representation, algorithms of calculation of available transitions of P systems to the next configuration, methods of P systems visualization, to reveal the points for effective paralleling (threading) of internal simulator processes. Then we plan to use

⋆ The authors acknowledge Dr. A.Alhazov for useful advices and fruitful discussions in the process of work over the software and the paper.
these methods, algorithms and developed code (sources) to simulator implementation for hybrid parallel processing using OpenMP and Cuda on a single PC in the cluster, and the MPI to involve all PC’s in the cluster.

The main distinction and advantage of the simulator presented here from the majority of already existing ones [5,6,8] is the totality of the following characteristics:

- work with string objects;
- work with nondeterministic P systems;
- user-friendly means for computation process control;
- a variety of visualization means to present P system configuration and evolving process.

2 Definition of P systems with String Replication

The formal definition of membrane P systems with string replication is given in [3]. A P system with string objects and input is a construct:

$$\Pi = (O, \Sigma, \mu, M_{l_1}, \ldots, M_{l_p}, R_{l_1}, \ldots, R_{l_p}, l_0),$$

where

- $O$ – a finite alphabet;
- $\Sigma$ – a sub-alphabet, $\Sigma \subseteq O$;
- $\mu$ – a membrane structure defined as a rooted tree with nodes labeled $1, \ldots, p$, the interior of each membrane defines a region;
- $M_{l_i}$ – multiset of strings, initially present in region $l_i$, $1 \leq i \leq p$;
- $R_{l_i}$ – set of rules of region $l_i$, $1 \leq i \leq p$;
- $l_{i_0}$ – label of input region, $1 \leq i_0 \leq p$.

A replication rule has the following structure:

$$a \rightarrow (u_1, t_1)||(u_2, t_2)||\ldots||(u_n, t_n),$$

where

- $a \in O^+$,
- $u_j \in O^*$, $1 \leq j \leq n$,
- $t_j \in \{\text{out, here}\} \cup \{\text{in}_{l_i} \mid 1 \leq i \leq p\}$, $1 \leq j \leq n$.

It is the string rewriting rule with string replication and target indication. Application of a rule $a \rightarrow (u_1, t_1)||(u_2, t_2)||\ldots||(u_n, t_n)$ from the set of rules $R_{l_k}$ transforms any string of the form $w_1aw_2$ from region $l_k$ into $n$ strings $w_1u_1w_2, w_1u_2w_2, \ldots, w_1u_nw_2$, where $w_1, w_2 \in O^*$. The resulting string $w_1u_jw_2$ should be sent to the region specified by $t_j$:

- if $t_j = \text{here}$, the resulting string remains in the region $l_k$;
- if $t_j = \text{out}$, the resulting string is sent out of the region associated with membrane with label $l_k$ to the region immediately outside;
- if \( t_j = in_{l_i} \), the resulting string is sent into the region associated with membrane with label \( l_i \), which has to be immediately nested into the region \( l_k \).

(If \( k = 1 \) we have the usual string rewriting rule with target indication.)

The initial configuration contains the input string(s) over \( \Sigma \) in region \( i_0 \) and the multisets of strings \( M_i \) in regions \( i \). The rules of the system are applied in parallel to all strings in the system. It may occur that several rules are applicable to a string. But really only one of them can be applied. The rule which will be applied, is determined in non-deterministic way. The computation consists in non-deterministic application of the rules to the strings in regions. If some string can be involved into the computing process, it has to be involved.

The computation halts when no rules are applicable. The result of the computation is the set of all words sent out of the outermost region into environment.

3 Representation of P system Description in Simulator Interface

We tried to provide in the interface a notation of P system, close to the conventional one in the theory of membrane computing, i.e. the signs of "\( \rightarrow \)" and "||", reserved words "here", "in", "out" for target indication in rule writing. But sometimes it occurs to be impossible. For example, to represent strings we use the concatenation symbol "," between alphabet objects without fail, although it is usually omitted in the theory. Moreover the strings are included in angle brackets "<" and ">".

The simulator is intended to serve the creative process of P system construction: when the researcher does not know from the very beginning the final variant of P system he needs, but invents and constructs it "on-the-run". So during the process of input of P system description the finite alphabet and sub-alphabet may replenish.

The simulator has conventionally-graphical interface for P system description input and editing (see Fig. 1). We say "Conventionally" – because not all elements of P system can be given by graphical means. The alphabet objects can be complex and consist of several keyboard symbols. They can have indexes as well. When the user inputs a string, rule or multiset, the interface guides him by widgets which significantly simplify the input process. The alphabets objects, labels and other units, which have already been input, are available for user by means of combo boxes or other control elements of the interface. The obligatory symbols (i.e. "," in string writing, \( \rightarrow \) in rule writing, etc.) are written (or prompted) by the simulator automatically.

When P system description is input by graphical interface, the corresponding XML format for this P system can be generated. This is intended for P system configurations saving, exchanging and transmission by simulator modules and advanced users. Fragments of XML file with P system configuration description appear in Fig. 2.
Fig. 1. A screenshot from the process of P system description input by graphical interface.

```xml
<Description strings with replication rules default configuration</Description>
<Parameters/>
<Membranes>
  <Multiplicity>0</Multiplicity>
</Membranes>
<Label>Environments</Label>
<ChamberNeutral>Chamber</Chamber>
<Rule />
<Source/>
<Target/>
</Info>
<Membranes>
  <Multiplicity>0</Multiplicity>
  <Multiplier />
</Membranes>
<Source/>
<Target/>
</Info>
<Rule>
  <ReplicationRule>
    <Multiplier>0</Multiplier>
    <Label>Neutral</Label>
  </ReplicationRule>
</Rule>
</Info>
</Description>
```

Fig. 2. Fragments of P system configuration description in XML format.
4 Visualization of P system Configuration

To discuss and demonstrate the simulator’s possibilities we have chosen an example of nondeterministic P system with string replication which the authors of [9] had constructed to prove that $RRP_4 - MAT \neq \emptyset$. The P system generates language $L$ which does not belong to $MAT$ and $L \cap a^* = \{a^{2n} | n \geq 1\}$.

Here $MAT$ is the family of languages generated by the matrix grammars without appearance checking, $RRP_4$ is the family of languages computed by non-extended replicated rewriting P systems of degree 4. For further details we refer to [9].

$$II = (V, V, \lambda_1 \lambda_2 \lambda_3 \lambda_4, 1, \emptyset, \{ca\}, \emptyset, \emptyset, R_1, R_2, R_3, R_4),$$

where

$$V = \{a, b, c, c', Z\},$$
$$R_1 = \{c' \rightarrow (\lambda, \text{out}), \ a \rightarrow Z, \ Z \rightarrow Z\},$$
$$R_2 = \{a \rightarrow \text{bb}, \ a \rightarrow (\text{bb, in})||(\text{bb, out}), \ c \rightarrow \{c', \text{out}\}\},$$
$$R_3 = \{b \rightarrow a, \ b \rightarrow (a, \text{in})||(a, \text{out})\},$$
$$R_4 = \{b \rightarrow Z, \ Z \rightarrow Z\}.$$

The screenshot of the simulator interface containing the description of this P system is presented at Fig. 3.

![Fig. 3. P system description by means of the simulator.](image_url)

In the simulator there are several possibilities to visualize P system configuration:

- by graphical representation of membrane structure as rectangles with labels and charges. The strings of multisets are written inside the rectangles (see middle panel of the interface in Fig. 3);
- by tree with nodes-membranes (see left panel "Current Configuration Tree" of the interface in Fig. 3). The tree reflects the membrane structure with its inner membranes in natural way. The membrane content and rules are represented as leaves. The detailed information on the membrane can be hidden or made visible by collapsing or expanding the respective node.

5 Available Transitions

The right panel "Available Transitions" (see Fig. 3) serves to show all possible variants of P system transitions from current configuration to the subsequent ones. Since in general, P systems are nondeterministic, there can be several variants of their transition to the subsequent configuration at any step of evolving. So the panel "Available Transitions" contains the list of these variants. Each variant consists of a set of complex elements. Each element indicates one rule and one string (to which the rule is applied) by the following way: membrane label; rule name; string; position in the string.

Each variant can be collapsed or expanded (to see the details of the variant) optionally. To watch which substrings of multiset strings will be changed if the variant (or a rule of the variant) is applied, the user can click the variant (or rule). Then the respective substring(s) will be highlighted by red color.

5.1 Computation Tree

To reflect the general picture of evolving process and calculated P system configurations dependence, there is the panel "Computation Tree" in the simulator interface. Each node of the tree represents some P system configuration. The initial P system configuration is a tree root. The tree structure shows the sequence of P system transitions from a configuration to another. Due to P system nondeterministic nature, every node (configuration) can have several children (possible subsequent configurations). So every tree branch represents one of the possible paths for P system evolving.

The Computation Tree helps the user to orientate in the "fantail" of the evolving process, to walk through it, to make steps backwards, to select the branch(es)/variant(s) for the next evolving, to apply selected branch(es)/variant(s), to ask the simulator to calculate the variants of transitions and subsequent configurations. Figure 4 gives just the idea of the Computation Tree appearance.

The Computation Tree grows during and along with the evolving process. A node appears in the Computation Tree when a possible variant of P system transition is calculated for parent node. Each node obtains its own standard name which reflects its place in the evolving path.

The appearance of the node shows to the user if the respective configuration is already calculated: the node name is written by bold type in this case, and by normal type otherwise. In the case when available transitions for a configuration are calculated, their number appears near the respective node name.
When the Halt configuration is achieved, this number is equal to zero. If the number of available transitions is absent, it means that available transitions for this configuration are not calculated yet.

Fig. 4. An example of "Computation tree" state for the chosen example of nondeterministic P system.

With the help of context menu (right click on the node in the tree) the user can:

- **see the standard name of configuration.** By default every configuration obtains "standard name" which reflects all the path from initial configuration to this one. For example, "0.2.1" means that the node is at the 2-nd level (each level is one step of calculation); in order to pass from the 0 level to the 1-st one, the second variant of transition was chosen; in order to pass from the first level to the 2-nd one, the first variant of transition was chosen.

- **change the name of configuration.** Before changing the "standard name" of configuration, it is considered as "current name" of configuration. Optionally the user can change the "standard name" of configuration. He can make it shorter, more convenient and meaningful. The changed name will become "current name" of configuration. The "current name" of configuration is seen in the computation tree. But the "standard name" of configuration is always available through the context menu. The user can reference to the configuration, using both the "standard" and the "current name" of configuration (for example, in order to search the configuration in the computation tree).

- **compute configuration.** This menu item is available only in the case when the configuration is not calculated yet. Thus, it is known which rules have to be applied to obtain the configuration from the parent one, but the rules are not applied yet. By this menu item these rules are applied, the configuration name in the computation tree turns bold, and is visualized in the left and middle panels with current configuration tree and current configuration.

- **compute available transitions.** By this menu item all possible transitions from given configuration will be calculated. In correspondence with them
the child-nodes will appear in computation tree for given node-configuration. "Standard names" will be generated for them. Since the respective configurations are not calculated yet, these names are written by normal type.

A state of the computation for the chosen example is shown in the Fig. 3:

- for the first branch "configuration 0, configuration 0.1, configuration 0.1.1, configuration 0.1.1.1" there were calculated three levels. The branch was ended by Halt.
- for the second branch "configuration 0, configuration 0.2" there was calculated only one level. And for the configuration 0.2 the possible transitions (4 variants) were calculated, but the configurations for these variants are not calculated yet. Hence their names are written by normal (not bold) type.
- for the third branch "configuration 0, configuration 0.3" there was calculated only one level too. The configuration 0.3 was not calculated. Hence the configuration name is written by normal (not bold) type and the number of possible transitions is not shown after configuration name.

6 Conclusion

The final goal is to develop tools for constructing simulators which will cover several types of P systems with string objects. So the simulator structure is organized in such a way, that only modules of available transitions calculation and rules application depend on P system type. At the same time the authors revealed the points for effective paralleling of simulation process.

References

A P–Lingua based Simulator for Spiking Neural P Systems


Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
{lmaciasr,perezh,mgarciaquismondo,lvalencia,marper,ariscosn}@us.es

Abstract. The research within the field of Spiking Neural P systems (SN P systems, for short) is focusing mainly in the study of the computational complexity and efficiency of this kind of systems. These devices have been shown capable of providing polynomial time solutions to computationally hard problems. In order to experimentally explore this computational power, it is necessary to develop software that provides simulation tools (simulators) for the existing variety of SN P systems. Such simulators allow us to carry out computations of solutions to computationally hard problems on certain instances. Within this trend, P-Lingua provides a standard language for the definition of P systems. As part of the same project, pLinguaCore library provides particular implementations of parsers and simulators for the models specified in P-Lingua. In this paper, an extension of the P-Lingua language to define SN P systems is presented, along with an upgrade of pLinguaCore including a parser and a new simulator for the variants of these systems included in the language.

1 Introduction

Spiking neural P systems were introduced in [9] in the framework of membrane computing [13] as a new class of computing devices which are inspired by the neurophysiological behaviour of neurons sending electrical impulses (spikes) along axons to other neurons. Since then, many computational properties of SN P systems have been studied; for example, it has been proven that they are Turing-complete when considered as number computing devices [9], when used as language generators [4, 2] and also when computing functions [15].

Investigations related to the possibility of solving computationally hard problems by using SN P systems were first proposed in [3]. The idea was to encode the instances of decision problems in a number of spikes which are placed in an arbitrarily large pre-computed system at the beginning of the computation. It was shown that the resulting SN P systems are able to solve the NP-complete
problem SAT (the satisfiability of propositional formulas expressed in conjunctive normal form) in a constant time. Slightly different solutions to SAT and 3-SAT by using SN P systems with pre-computed resources were considered in [10]; here the encoding of an instance of the given problem is introduced into the pre-computed resources in a polynomial number of steps, while the truth values are assigned to the Boolean variables of the formula and the satisfiability of the clauses is checked. The answer associated with the instance of the problem is thus computed in a polynomial time. Finally, very simple semi-uniform and uniform solutions to the numerical \textbf{NP}-complete problem \textit{Subset Sum} – by using SN P systems with exponential size pre-computed resources – have been presented in [11]. All the systems constructed above work in a deterministic way.

In [12], neuron division and budding were introduced into the framework of SN P systems in order to enhance the efficiency of these systems. The biological motivation of these features was founded on recent discoveries in neurobiology related to neural stem cells [6]. Neural stem cells persist throughout life within central nervous system in the adult mammalian brain, and this ensures a life-long contribution of new neurons to self-renewing nervous system with about 30000 new neurons being produced every day.

In this paper, a simulator of SN P systems with neuron division and budding is presented based on P-Lingua. P-Lingua is a programming language to define P systems [5, 7, 22], that comes together with a Java library providing several services; (e.g., parsers for input files and built–in simulators). In this paper we present a new release of P-Lingua. One of the innovations is an extension of the previous syntax in order to define SN P systems with neuron division and budding. Furthermore, the library has been updated to handle P-Lingua input files defining SN P systems. Finally, a new built–in simulator has been added to the library in order to simulate computations of such new models.

The paper is structured as follows. In Section 2 we introduce some definitions about SN P systems with neuron division and budding. Section 3 describes the extensions for P–Lingua programming language in order to support that kind of SN P systems. In Section 4, we introduce the simulator for these P systems presented in this paper, including the simulation algorithm. Section 5 illustrates the use of the simulator through a case study, the SAT problem. Section 6 is devoted to performance considerations. Finally, conclusions and future work are discussed in Section 7.

2 Preliminaries

This section is based on an extract from [12]. In that article, a formal description of SN P systems with neuron division and budding is outlined. Our simulator is modelled after that article, rather than the one [9] that introduced this variety.
SN P systems can be considered a variant of P systems, corresponding to a shift from cell-like to neural-like architectures. In these systems the cells (also called neurons) are placed in the nodes of a directed graph, called the synapse graph. The contents of each neuron consist of a number of copies of a single object type, called the spike. Every cell may also contain a number of firing and forgetting rules. Firing rules allow a neuron to send information to other neurons in the form of electrical impulses (also called spikes) which are accumulated at the target cell. The applicability of each rule is determined by checking the contents of the neuron against a regular set associated with the rule. In each time unit, if a neuron can use one of its rules, then one of such rules must be used. If two or more rules could be applied, then only one of them is non-deterministically chosen. Thus, the rules are used in a sequential way in each neuron, but neurons function in parallel with each other. Note that, as it usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, and hence the functioning of the system is synchronized. When a cell sends out spikes it becomes “closed” (inactive) for a specified period of time. During this period, the neuron does not accept new inputs and cannot “fire” (that is, emit spikes). Furthermore, SN P systems associate a delay parameter to each rule which occurs in the system. If no firing rule can be applied in a neuron, then there may be the possibility to apply a forgetting rule, that removes from the neuron a predefined number of spikes.

The structure of SN P systems (that is, the synapse graph) introduced in [9] is static. Nevertheless, our simulator supports dynamical neural structures. These structures may evolve throughout computations by means of division and budding rules [12].

Formally, a spiking neural P system with neuron division and budding of degree $m \geq 1$ is a construct of the form:

$$\Pi = (O, H, syn, n_1, \ldots, n_m, R, in, out),$$

where:

1. $m \geq 1$ (the initial degree of the system);
2. $O = \{a\}$ is the singleton alphabet ($a$ is called spike);
3. $H$ is a finite set of labels for neurons;
4. $syn \subseteq H \times H$ is a synapse dictionary, with $(i, i) \notin syn$ for $i \in H$;
5. $n_i \geq 0$ is the initial number of spikes contained in neuron $i$, $i \in \{1, 2, \ldots, m\}$;
6. $R$ is a finite set of developmental rules, of the following forms:
   (1) extended firing (also called spiking) rule $[E/a^c \rightarrow a^p, d]_i$, where $i \in H$, $E$ is a regular expression over $a$, and $c \geq 1$, $p \geq 0$, $d \geq 0$, with the restriction $c \geq p$;
   (2) neuron division rule $[E]_i \rightarrow []_j \parallel []_k$, where $E$ is a regular expression and $i, j, k \in H$;

of P systems, as our simulator supports extended rules.
(3) neuron budding rule \([E]_i \rightarrow []_i/[]_j\), where \(E\) is a regular expression and \(i, j \in H\);

7. \( \text{in}, \text{out} \in H \) indicate the input and the output neurons of \( \Pi \).

Note that the above definition is slightly different from the usual one found in the literature, where the neurons initially present in the system are explicitly listed as \( \sigma_i = (n_i, R_i) \), where \(1 \leq i \leq m\) and \(R_i\) are the rules associated with neuron with label \(i\). In what follows we will refer to neuron with label \(i \in H\) also denoting it with \(\sigma_i\).

If an extended firing rule \([E/a^c \rightarrow a^p; d]_i\) has \(E = a^c\), then we will write it in the simplified form \([a^c \rightarrow a^p; d]_i\); similarly, if a rule \([E/a^c \rightarrow a^p; d]_i\) has \(d = 0\), then we can simply write it as \([E/a^c \rightarrow a^p]_i\); hence, if a rule \([E/a^c \rightarrow a^p; d]_i\) has \(E = a^c\) and \(d = 0\), then we can write \([a^c \rightarrow a^p]_i\). A rule \([E/a^c \rightarrow a^p]\), with \(p = 0\) is written in the form \([E/a^c \rightarrow \lambda]_i\) and is called extended forgetting rule. Rules of the types \([E/a^c \rightarrow a; d]_i\) and \([a^c \rightarrow \lambda]_i\), are said to be standard.

In addition to spiking rules, our simulator supports neuron division and neuron budding rules. Basically, division rules create a pair of new membranes out of a previously existing one (that disappears). These membranes are placed in parallel. Membranes created by means of division rules inherit the synapses going into and out of the original membrane. On the other hand, budding rules create a new membrane while preserving the original one. This new membrane is placed serially following the original one. Furthermore, the new membrane created by means of budding rules inherits the outgoing synapses of the parent membrane. These synapses are transferred from the parent to the child membrane. In addition, a synapse from the original membrane to the newly created membrane is established. Thus, after the application of division rules two new membranes with (potentially) new labels are created. The original one disappears. On the other hand, after the application of budding rules only a new membrane with a new (potentially) label is created, preserving the original membrane. In both cases additional synapses are added connecting the child membranes with pre-existent membranes according to the rules dictated by the synapse dictionary. As a result of the simulator supporting both division and budding rules, the membrane structure of the system is able to evolve dynamically along the computation steps.

A more thorough description of the semantics of SN P systems can be found in [12] and in [18].

The configuration of the system is described by the topological structure of the system, the number of spikes associated with each neuron, and the state of each neuron (open or closed). Using the rules as described above, one can define transitions among configurations. Any sequence of transitions starting in the initial configuration is called a computation. A computation halts if it reaches a configuration where all neurons are open and no rule can be used.
Traditionally, the input of an SN P system used in the accepting mode is provided in the form of a spike train, a sequence of steps when one spike or no spike enters the input neuron. Thus, we need several spikes at a time to come into the system via the input neuron, i.e., we consider “generalized spike trains” instead, written in the form \( a^{i_1} \cdot a^{i_2} \cdots a^{i_r} \), where \( r \geq 1 \), \( i_j \geq 0 \) for each \( 1 \leq j \leq r \). The meaning is that \( i_j \) spikes are introduced in neuron \( \sigma_{in} \) in step \( j \) (all these \( i_j \) spikes are provided at the same time).

3 P-Lingua syntax for SN P systems

This section is devoted to show the extended P-Lingua syntax that covers the SN P systems considered in the proposed P-Lingua language revision. The following paragraphs describe the new features incorporated in order to support SN P systems. These features include different simulation modes, an initial membrane structure (which consists of the initial set of membranes and a collection of synapses), a simulation input spike train and the explicit (and optional) specification of input and output membranes. These features also include language-related items, such as regular expressions and new reserved words.

3.1 Regular expressions

Regular expressions have been introduced. These regular expressions are a subset of those defined according to the Java package `java.util.regex` formally specified at [20]. This subset is defined by combining the following symbols as established in the syntax just mentioned:

\[ 'a', '(', ')', '[', ']', '{', '}', ',', '^', '*', '+', '?' \]

The purpose of the inclusion of regular expression in this version of P-Lingua is to specify the regular expressions associated to the rules of SN P systems.

Additional information related to the Java language can be found at [21]. We remark that these regular expressions are not checked by the parser. Instead, they are piped directly into the simulator. Then, the simulator performs the parsing. Regular expressions \( E \) are written double-quoted: \"E\". Also, in this version the membrane labels can accept parameters built on integer expressions. In addition, no new arithmetic operator has been introduced in this revision.

3.2 Model specification

In this version of P-Lingua, a SN P system specification must define an initial membrane structure and a set of rules. The membrane structure is composed of a set of membranes joined by synapses. These synapses are specified as a set of connections. If the SN P system considers division and budding rules, then a
Apart from the initial configuration and the set of rules, in this version of P-Lingua the user can also define an input membrane and a set of output membranes. In addition, the user can also define simulation parameters. These parameters include simulation modes and spike trains.

P-Lingua files defining SN P systems must begin with the following sentence:

\texttt{@model<spiking_psystems>}

Additionally, \texttt{@model<spiking_psystems>} can be followed by a pair of sentences in order to specify the simulation modes to be used. These modes determine the semantics in which the simulation is performed as described in [8]. The sentences are:

\begin{itemize}
  \item \texttt{@masynch = v1;}
    \begin{itemize}
      \item $v1 \in \{0, 1, 2\}$. If this sentence is not present, then $v1$ defaults to 0.
    \end{itemize}
  \end{itemize}

These values denote the following modes:

0: Synchronous (standard) mode.
1: Asynchronous unbounded mode.
2: Asynchronous bounded mode.

Let us consider a membrane structure with $N$ membranes. If \texttt{@masynch} is set to 2 then the next sentence can be used to express the valid halting configuration:

\texttt{@mvalid = (m1,n1), (m2,n2),..., (mN,nN);}

where, for each integer $i \in [1, \ldots, N]$:

\begin{itemize}
  \item $m_i$ is a membrane label in the SN P system.
  \item $n_i$ is an integer expression which specifies the number of spikes contained in $m_i$ at the end of the computation.
\end{itemize}

If the sentence is not used then every halting configuration is considered valid. Also if the sentence is used when \texttt{@masynch} is not set to 2, it would be ignored.

\begin{itemize}
  \item \texttt{@mseq = v2;}
    \begin{itemize}
      \item $v2 \in \{0, \ldots, 5\}$. If this sentence is not present, then $v2$ defaults to 0.
    \end{itemize}
\end{itemize}

These values denote the following modes:

0: parallel (standard) mode.
1: standard sequential mode.
2: max pseudo-sequential mode.
3: max sequential mode.
4: min pseudo-sequential mode.
5: min sequential mode.
Let us consider an initial membrane structure of a SN P system with \( N \) membranes and \( M \) synapses. In this version of P-Lingua, in order to define that initial membrane structure, the following sentence must be written:

\[ \text{@mu} = m_1, m_2, \ldots, m_N; \]

where, for each integer \( i \in [1, \ldots, N] \), \( m_i \) is the label of membrane \( i \). The label \textit{environment} cannot be used. Given an initial membrane structure, in order to define the connections between the membranes, the following sentence must be written:

\[ \text{@marcs} = \text{arc}_1, \text{arc}_2, \ldots, \text{arc}_M; \]

where, for each integer \( i \in [1, \ldots, M] \), \( \text{arc}_i = (m_{k_i}, m_{l_i}) \), \( m_{k_i} \) and \( m_{l_i} \) being two membrane labels of an SN P system configuration and \( m_{k_i} \neq m_{l_i} \).

Let us consider a dictionary. In this version of P-Lingua, in order to specify that synapse dictionary the following optional sentence must be written:

\[ \text{@mdict} = e_1, e_2, \ldots, e_D; \]

where \( D \) is the number of entries of the dictionary and, for each integer \( p \in [1, \ldots, D] \), \( e_p = (l_i, l_j) \), \( l_i \) and \( l_j \) are two different labels and \( l_i \neq l_j \).

If the SN P system specification contains division or budding rules the dictionary is mandatory. Besides, an implicit dictionary is built from the parameter \text{@marcs}. This dictionary is extended by the “explicit” dictionary defined by the sentence \text{@mdict}.

Let us consider the previously defined SN P system schema. Let us consider an input membrane. In this version of P-Lingua, that input membrane is optional, i.e., P-Lingua allows the specification of SN P systems with no input membrane. That input membrane receives an input spike train throughout time. Let us consider that spike train. In this version of P-Lingua, in order to specify the (optional) input membrane of the SN P system, the following sentence may be written:

\[ \text{@min} = m; \]

where \( m \) is the label of a membrane existing in the initial SN P system membrane structure.

Let us consider an input spike train for a SN P system with an input membrane defined in the schema mentioned above. This train consists of \( S \) steps. In this version of P-Lingua, in order to specify this input spike train, the following sentence must be written:

\[ \text{@minst} = r_1, r_2, \ldots, r_S; \]
where, for each integer \( i \in [1, \ldots, S] \), \( r_i = (i, a_i) \), \( i \) and \( a_i \) being two integer expressions, \( i>1 \) and \( a_i \geq 0 \). The parameter \( i \) denotes the step of the computation calculated by the simulator while the parameter \( a_i \) denotes the number of spikes that are introduced in the input membrane at step \( i \). When the pair \( r_j = (j, a_j) \) is undefined for some step \( j \) then the simulator assumes that zero spikes are introduced in the system at that step.

Let us consider the previously defined SN P system schema. Let us consider a P system defined in this schema. Let us consider an output membrane. In this version of P-Lingua, the output membrane is optional, i.e., P-Lingua allows the specification of SN P systems with no output membrane. In this version of P-Lingua, in order to specify the (optional) output membranes of the system, the following sentence may be written:

\[
@\text{mout} = o_1, o_2, \ldots, o_L;
\]

where, for each integer \( i \in [1, \ldots, L] \), \( o_i \) denotes a membrane label of the SN P system.

### 3.3 Sample of the first few lines of a P-Lingua SN P system definition file

The following paragraph shows the first few lines of a P-Lingua SN P system definition file. These lines cover examples of the previously defined syntax.

```p-lingua
@model<spiking_psystems>

@masynch = 2;
@mvalid = (1, 3), (2, 6), (3, 4);
@mseq = 2;
@mu = 1,2,3;
@marcs = (1,2), (1,3);
@mdict = (1,d),(2,f);
@min = 1;
@minst = (1,3), (5,4), (8,2);
@mout = 1,2;
```

### 3.4 Definition of multisets

Initial multisets of objects for neurons can be defined in the same way as initial multisets of objects for cell-like membranes, with the restriction that only the \( a \) object may be used.

For instance:
3.5 Definition of rules

If a P-Lingua file begins with the @model<spiking psystems> sentence, then four types of rules can be defined:

(1) **Firing rules**, that can be specified in the following ways:
   - \([a*c]'h \rightarrow [a*p]'h \ "e" :: d\);
   - \([a*c] \rightarrow a*p]'h \ "e" :: d\);

(2) **Forgetting rules**, that can be specified in the following ways:
   - \([a*c]'h \rightarrow [#]'h \ "e" :: d\);
   - \([a*c] \rightarrow #]h \ "e" :: d\);

(3) **Neuron division rules**, that can be specified in the following way:
   \([]'i \rightarrow [#]'j || [#]'k \ "e"\);

(4) **Neuron budding rules**, that can be specified in the following way:
   \([]'i \rightarrow [#]'i / [#]'j \ "e"\);

where \(h\), \(i\), \(j\), and \(k\) are membrane labels of the SN P system described, \(c\), \(p\) and \(d\) are integer expressions which satisfy that \(c \geq 1\), \(c \geq p\) and \(d \geq 0\), and \(e\) is a regular expression over \{a\}.

For firing and forgetting rules, both \(d\) and \"e\" are optional with \(d\) defaulting to 0. In forgetting rules \(d\) is always set to 0. When \(e\) is not present in the rule, then \(e\) defaults to the left hand side of the rule.

Division and budding rules have no delays, so \(d\) is not used, but the regular expression \(e\) is mandatory.

For instance, the following rules are valid spiking rules in P-Lingua:

- \([a*3]'1 \rightarrow [a*2]'1 \ "a^3" :: 3\);
- \([a*3] \rightarrow a*2]'1 \ "a^2" :: 6\);
- \([a*3] \rightarrow #]'1 \ "a+a^2" :: 3\);
- \([a*3] \rightarrow #]'1 \ "a*a" :: 5\);

Also, the following rules are valid division and budding rules in P-Lingua:

- \([]'1 \rightarrow [](2 || [](3 + a^n)\);
- \([]'1 \rightarrow [](1 / [](2 (a^3)a)\);

Recall that in P-Lingua, the \# symbol is optional (it can be omitted), for instance the following rules:

- \([]'1 \rightarrow #]'2 || [#]'3 \ "a^n"\);
3.6 Definition of the output

In order to specify additional output results to be shown to the user after the computation halts (when it halts), the following sentences may be included:

@moutres_binary;
@moutres_natural(k,strong,alternate);
@moutres_summatories;

where all of them are optional.

When the @moutres_binary parameter is specified then the output is shown as a binary spike train. If more than one output membrane was defined then a binary output spike train is shown for each one of them. The binary spike train for a given output membrane $o_j$ is a binary sequence $b_1, b_2, ..., b_N$ with $b_i = 0$ if and only if $o_j$ sends no spikes to the environment at computation step $i$, and 1 otherwise.

When the @moutres_natural parameter is specified then a natural output spike train is shown to the user. If more than one output membrane was defined then a train is shown for each output membrane. For a thorough explanation please refer to [14]. If natural($k$,strong,alternate) is specified, then $k$ is an integer expression with $k \geq 2$ and strong and alternate take boolean values (false or true).

When the @moutres_summatories parameter is specified then the sum of the spikes sent to the environment for each output membrane, (i.e., the contents of the environment) is shown as output.
3.7 Reserved words

The set of reserved words has been updated in the current syntax of the P-Lingua language by adding the following text strings:

@masynch, @mseq, @mvalid, @marcs, @mdict, @min, @minst, @mout, |, @moutres_binary, @moutres_natural, @moutres_summatories

The inclusion of these reserved words is necessary in order to include the new features of this version of P-Lingua. These new features are explained above.

4 A simulator software for SN P systems

In [7], a Java library called $pLinguaCore$ was presented under GPL license. It includes parsers to handle input files and built-in simulators to generate P system computations. It can export several output file formats to represent P systems. It is not a closed product because developers with knowledge of Java can add new components to the library. In this paper, $pLinguaCore$ has been upgraded to support SN P systems. Now, the library is able to handle input P-Lingua files which define SN P systems and it includes a new built-in simulator in order to simulate SN P system computations. The current version of the library can be downloaded from http://www.p-lingua.org.

The simulation algorithm described below generates one possible computation for a SN P system with an initial configuration $C_0$ containing $n$ membranes $m_1, \ldots, m_n$. Recall that when working with recognizer P systems all computations yield the same answer (confluence).

The simulation algorithm is structured is six steps:

I. Initialization
In this step the data structures needed to conduct the simulation are initialised.

II. Selection of rules
In this step the set of rules to be executed in the current step is calculated.

III. Build execution sets
In this step the rules to be executed are split into different sets, according to their kind.

IV. Execute division and budding rules
In this step division and budding rules are executed. The execution is performed in two phases: In the first one, new neurons are calculated out of existing neurons by applying budding and division rules. In the second one additional synapses are introduced according to the synapse dictionary.
V. Execute spiking rules
In this step execution of spiking rules is performed.

VI. Ending
In this step the current configuration is updated with the configuration newly calculated and the halting condition is checked (no rules are applicable).

In what follows, the simulation algorithm is described.

I. Initialization
1. Let $C_t$ be the current configuration
2. Let $M_{sel} ≡ \emptyset$ be a set of membranes who are susceptible of executing a rule in the current computation step
3. Let $m_0$ be a virtual membrane (with label 0) representing the environment

II. Selection of rules
1. Each membrane $m_i$ stores the following elements:
   - last rule $r_i$ selected to be executed in a previous step for that membrane (initially none)
   - an integer decreasing-only counter $d_i$, that stores the number of steps left for the membrane to open and fire in case $r_i$ is a firing rule (initially zero).

   For each membrane $m_i$, do
   (a) If $m_i$ is closed as a result of being involved in the execution of a budding or division rule, then open $m_i$ (let $d_i = 0$) and clear its rule $r_i$
   (b) If $m_i$ is closed as a result of being involved in the execution of a firing rule (thus $r_i$ is a firing rule) then
      i. Decrease the counter $d_i$
      ii. Add $m_i$ to $M_{sel}$
   iii. Go to process the next membrane
   (c) Let $S_i ≡ \emptyset$ be the set of possible rules to be executed over $m_i$
   (d) For each rule $r_j$ with label $j$ do
      i. If $r_j$ is active and can be executed over $m_i$ then add $r_j$ to $S_i$
   (e) If $S_i$ is empty then go to process the next membrane
   (f) Select non deterministically a rule $r_k$ from $S_i$
   (g) Set $r_k$ as the new selected rule for $m_i$
   (h) If $r_k$ is a firing rule, update the counter $d$ accordingly
      (i) Add $m_i$ to $M_{sel}$
   (j) Clear $S_i$

2. If $M_{sel}$ is not empty and the simulator operates in sequential mode then
   (a) Select a membrane $m_s$ from $M_{sel}$ according to the sequential mode
   (b) Clear $M_{sel}$
   (c) Add $m_s$ to $M_{sel}$

III. Build execution sets
1. Let $\text{Division} \equiv \emptyset$ be the set that stores the membranes having a division rule selected to be executed in the current step.

2. Let $\text{Budding} \equiv \emptyset$ be the set that stores the membranes having a budding rule selected to be executed in the current step.

3. Let $\text{Spiking} \equiv \emptyset$ be the set that stores the membranes having a spiking rule selected to be executed in the current step (or susceptible to be executed in the case of firing rules with delays).

4. For each membrane $m_i$ from $M_{set}$ do:
   (a) Let $r_i$ be the selected rule for $m_i$.
   (b) If $r_i$ is a division rule then add $m_i$ to $\text{Division}$.
   (c) If $r_i$ is a budding rule then add $m_i$ to $\text{Budding}$.
   (d) If $r_i$ is a spiking rule then add $m_i$ to $\text{Spiking}$.

IV. Execute division and budding rules

1. Let $\text{Div} \equiv \emptyset$ be the set that stores the membranes that are generated as a result of applying a division rule in the current step.

2. Let $\text{Bud} \equiv \emptyset$ be the set that stores the membranes that are generated as a result of applying a budding rule in the current step.

3. For each membrane $m_i$ from $\text{Division}$ do:
   (a) If the simulator operates in asynchronous mode then
      i. Determine non deterministically if the rule has to be executed;
      ii. If the rule does not have to be executed then go to process the next membrane.
   (b) Let $r_i$ be the selected rule for $m_i$: $[E]_i \rightarrow [\emptyset | \emptyset | \emptyset ]_i$.
   (c) Relabel $m_i$ with the $j$ label, thus from now on we refer to $m_j$.
   (d) Create a new membrane $m_k$ and close it.
   (e) For each incoming edge from some membrane $m_p$ to $m_j$ create a new edge from $m_p$ to $m_k$.
   (f) For each outgoing edge from $m_j$ to some membrane $m_p$ create a new edge from $m_k$ to $m_p$.
   (g) Add $m_j$ and $m_k$ to $\text{Div}$.

4. For each membrane $m_i$ from $\text{Budding}$ do:
   (a) If the simulator operates in asynchronous mode then
      i. Determine non deterministically if the rule has to be executed;
      ii. If the rule has not to be executed then go to process the next membrane.
   (b) Let $r_i$ the selected rule for $m_i$: $[E]_i \rightarrow [\emptyset | \emptyset | \emptyset ]_i$.
   (c) Create a new membrane $m_j$ and close it.
   (d) For each outgoing edge from $m_i$ to some membrane $m_p$ do:
      i. Create a new edge from $m_i$ to $m_p$.
      ii. Remove the edge from $m_i$ to $m_p$.
   (e) Create a new edge from $m_i$ to $m_j$.
   (f) Add $m_j$ to $\text{Bud}$.

5. For each membrane $m_i$ from $\text{Div}$ create new edges involving $m_i$ according to the synapse dictionary if necessary.

6. For each membrane $m_i$ from $\text{Bud}$ create new edges involving $m_i$ according to the synapse dictionary if necessary.
V. Execute spiking rules

1. For each membrane $m_i$ from Spiking do
   (a) If $m_i$ is closed then go to process the next membrane
   (b) If the simulator operates in asynchronous mode then
      i. Determine non-deterministically if the rule has to be executed
      ii. If the rule does not have to be executed then go to process the next membrane
   (c) Let $r_i$ be the selected rule for $m_i$
   (d) If $r_i$ is a firing rule of the form $[E/a_c \rightarrow a^p; d]_i$ then
      i. Remove $c$ spikes from the multiset of $m_i$
      ii. For each membrane $m_j$ connected to $m_i$ by an edge going from $m_i$ to $m_j$, add $p$ spikes to the multiset of $m_j$ if and only if $m_j$ is open
   (e) If $r_i$ is a forgetting rule of the form $[E/a_c \rightarrow \lambda]_i$ then remove $c$ spikes from the multiset of $m_i$

VI. Ending

1. Let $C_{t+1} = C_t$
2. If $M_{sel}$ is not empty then goto I

Before going on, it is important to note that we assume the defined P system to be free of syntax errors that could lead to an incorrect computation, since the P–Lingua parser checks for any possible programming errors.

5 An example: a family of SN P systems solving SAT

In order to illustrate the operation of the simulator, a uniform and efficient solution to SAT by means of a family of SN P systems with spiking, neuron division and budding rules is described in this section.

We will start considering the description of the family in terms of its theoretical definition, then a P–Lingua source code implementing the solution will be shown and finally the simulation results will be presented.

5.1 Description of the Family

As mentioned above, a uniform and efficient solution to SAT by means of a family of SN P systems with spiking, neuron division and budding rules is described in what follows. This solution has already been presented in [12].

Let us consider a propositional formula $\text{SAT}(n, m) = C_1 \land \ldots \land C_m$ over $n$ variables $x_1 \ldots x_n$, consisting of $m$ clauses $C_j = y_{j,1} \lor \ldots \lor y_{j,k_j}$, $1 \leq j \leq m$, where $y_{j,i} \in \{x_l, \neg x_l | 1 \leq l \leq n\}$, $1 \leq i \leq k_j$. Without loss of generality, we may assume that no clause contains two occurrences of some $x_i$ or two occurrences of some $\neg x_i$ (the formula is not redundant at the level of clauses), or both $x_i$ and $\neg x_i$ (otherwise such a clause is trivially satisfiable, hence can be removed).
Because the construction is uniform, we need a way to encode any given instance $\gamma_{n,m}$ of SAT$(n,m)$. Each clause $C_i$ of $\gamma_{n,m}$ can be seen as a disjunction of at most $n$ literals, and thus for each $j \in \{1, 2, \ldots, n\}$ either $x_j$ occurs in $C_i$, or $\neg x_j$ occurs, or none of them occurs. In order to distinguish these three situations we define the spike variables $\alpha_{i,j}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$, as variables whose values are amounts of spikes; we assign to them the following values:

$$\alpha_{i,j} = \begin{cases} a, & \text{if } x_j \text{ occurs in } C_i; \\ a^2, & \text{if } \neg x_j \text{ occurs in } C_i; \\ a^b, & \text{otherwise.} \end{cases}$$

In this way, clause $C_i$ will be represented by the sequence $\alpha_{i,1} \cdot \alpha_{i,2} \cdot \ldots \cdot \alpha_{i,n}$ of spike variables; in order to represent the entire formula $\gamma_{n,m}$ we just concatenate the representations of the single clauses, thus obtaining the generalized spike train $\alpha_{1,1} \cdot \alpha_{1,2} \cdot \ldots \cdot \alpha_{1,n} \cdot \alpha_{2,1} \cdot \alpha_{2,2} \cdot \ldots \cdot \alpha_{2,n} \cdot \ldots \cdot \alpha_{m,1} \cdot \alpha_{m,2} \cdot \ldots \cdot \alpha_{m,n}$. As an example, the representation of $\gamma_{3,2} = (x_1 \lor \neg x_2) \land (x_1 \lor x_3)$ is the sequence $a \cdot a^2 \cdot a^b \cdot a \cdot a^b \cdot a$. In order to let the systems have enough time to generate necessary workspace before computing the instances of SAT$(n,m)$, a spiking train $(a^b)^{2n}$ is added at the beginning of the input. In general, for any given instance $\gamma_{n,m}$ of SAT$(n,m)$, the encoding sequence is $\text{cod}(\gamma_{n,m}) = (a^b)^{2n} \cdot \alpha_{1,1} \cdot \alpha_{1,2} \cdot \ldots \cdot \alpha_{1,n} \cdot \alpha_{2,1} \cdot \alpha_{2,2} \cdot \ldots \cdot \alpha_{2,n} \cdot \ldots \cdot \alpha_{m,1} \cdot \alpha_{m,2} \cdot \ldots \cdot \alpha_{m,n}$.

For each $n, m \in \mathbb{N}$, we construct

$$H((n,m)) = (O, H, \text{syn}, n_1, \ldots, n_q, R, \text{in}, \text{out}),$$

with the following components:

The initial degree of the system is $q = 4n + 7$;

$O = \{a\}$;

$H = \{m, \text{out}, d, cl\} \cup \{d_i \mid i = 0, 1, \ldots, n\}$
$\cup \{C_{x_i} \mid i = 1, 2, \ldots, n\} \cup \{C_{x_0} \mid i = 1, 2, \ldots, n\}$
$\cup \{C_{x_1} \mid i = 1, 2, \ldots, n\} \cup \{t_i \mid i = 1, 2, \ldots, n\}$
$\cup \{f_i \mid i = 1, 2, \ldots, n\} \cup \{0, 1, 2, 3\}$;

$\text{syn} = \{\{(d_i, d_{i+1}) \mid i = 0, 1, \ldots, n - 1\} \cup \{(d_n, d_1)\}\}$
$\cup \{(m, C_{x_i}) \mid i = 1, 2, \ldots, n\} \cup \{(d_i, C_{x_i}) \mid i = 1, 2, \ldots, n\}$
$\cup \{\{(C_{x_i}, C_{x_0}) \mid i = 1, 2, \ldots, n\} \cup \{(C_{x_i}, C_{x_1}) \mid i = 1, 2, \ldots, n\}$
$\cup \{(i + 1, i) \mid i = 0, 1, 2\} \cup \{(1, 2), (0, \text{out})\}\}$
$\cup \{(C_{x_1}, t_i) \mid i = 1, 2, \ldots, n\} \cup \{(C_{x_0}, f_i) \mid i = 1, 2, \ldots, n\}$;

$n_{d_0} = n_0 = n_2 = n_3 = 1, n_{d_1} = 6$, and there is no spike in the other neurons;

$R$ is the following set of rules:
(1) **spiking rules**:

$[a \rightarrow a]_{i,n}$

$[a^2 \rightarrow a^2]_{i,n}$.
Thus we are resolving \( \varphi \) as the instance of \( \text{SAT} \) that we try to solve:

\[
\varphi = (x_1 \lor x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor x_2) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor \neg x_3 \lor x_4) \land \\
(\neg x_1 \lor x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor x_4) \land (\neg x_1 \lor x_2 \lor x_3 \lor x_4)
\]

Thus we are resolving \( \text{SAT}(n, m) \) with \( n = 4 \) and \( m = 8 \). Of course, the \texttt{main} module can be easily modified in order to define any other P system of the family.

The source code is structured as follows:
1. Module `main()` that defines a SN-P system solving the SAT problem for the formula described above with 4 variables and 8 clauses. Firstly, it calls the module `spiking_init_conf(n)` for $n \equiv 4$. Secondly, it calls the module `spiking_rules(n,m)` for $(n, m) \equiv (4,8)$.

2. Module `spiking_init_conf(n)` that defines the initial configuration of a SN-P system solving the SAT problem for any instance with $n$ variables. Furthermore, this module defines the train of spikes which encode the $\varphi$ formula. This train is defined according to what we stated above.

3. Module `spiking_rules(n,m)` that defines the spiking rules of the SN-P system for any instance with $n$ variables and $m$ clauses.

4. Module `neuron_division_rules(n)` that defines the division rules of the SN-P system for any instance with $n$ variables.

5. Module `neuron_budding_rules(n)` that defines the budding rules of the SN-P system for any instance with $n$ variables.

The source code for the SN-P system defined above can be found in the URL: http://www.p-lingua.org/examples/sat_SNPSys.pli. In what follows we will show its most significant part.

```plaintext
@model<spiking_psystems>
// Encoding module main()
def main()
{
    call spiking_init_conf(4);
    call spiking_rules(4,8);
    call neuron_division_rules(4);
    call neuron_budding_rules(4);
}

// Encoding module spiking_init_conf()

def spiking_init_conf(n)
{

    // Encoding initial membranes
    @mu = in, out;
    @mu += 0,1,2,3;
    @mu += d{i} : 0<=i<=n;
    @mu += Cx{i} : 1<=i<=n;
    @mu += Cx{i,0} : 1<=i<=n;
    @mu += Cx{i,1} : 1<=i<=n;

    // Encoding initial membrane spikes
```
@ms(d{0}) = a;
@ms(0) = a;
@ms(2) = a;
@ms(3) = a;
@ms(d{1}) = a*6;

// Encoding initial synapse graph (also updating synapse dictionary)

@marcs = (d{i},d{i+1}):0<=i<=n-1;
@marcs += (d{n},d{1});
@marcs += (in,Cx{i}):1<=i<=n;
@marcs += (d{i},Cx{i}):1<=i<=n;
@marcs += (Cx{i},Cx{i,0}):1<=i<=n;
@marcs += (Cx{i},Cx{i,1}):1<=i<=n;
@marcs += ({i+1},{i}):0<=i<=2;
@marcs += (1,2);
@marcs += (0,out);

// Encoding additional synapse dictionary updating

@mdict = (Cx{i,1},t{i}):1<=i<=n;
@mdict += (Cx{i,0},f{i}):1<=i<=n;

// Encoding input neuron

@min = in;

// Encoding input formula spike train

// Encoding first clause

@minst = (9,1);
@minst += (10,1);
@minst += (11,2);
@minst += (12,1);
...

// Encoding last clause

@minst += (37,2);
@minst += (38,1);
@minst += (39,1);
@minst += (40,1);
// Encoding output neuron
@mout = out;
}

// Encoding module spiking_rules()

def spiking_rules(n,m)
{
    [a --> a]'in;
    [a*2 --> a*2]'in;
    [a --> a]'d(0) : 2*n+n*m;
    [a*4 --> a*4]'d(i) : i<=i<=n;
    [a*5 --> #]'d(1);
    [a*6 --> a*4]'d(i) : 2*n+1;
    [a --> #]'Cx{i} : 1<>i<>n;
    [a*2 --> #]'Cx{i} : 1<>i<>n;
    [a*4 --> #]'Cx{i} : 1<>i<>n;
    [a*5 --> a*5]'Cx{i} : n-i : 1<>i<>n;
    [a*6 --> a*6]'Cx{i} : n-i : 1<>i<>n;
    [a*5 --> a*4]'Cx(i,1) : 1<>i<>n;
    [a*6 --> #]'Cx{i,0} : 1<>i<>n;
    [a*5 --> #]'Cx{i,0} : 1<>i<>n;
    [a*6 --> a*4]'Cx{i,0} : 1<>i<>n;
    [a --> a]'t{i} "(a{4})+" : 1<>i<>n;
    [a --> a]'f{i} "(a{4})+" : 1<>i<>n;
    [a*(4*k-1) --> #]'t{i} : 1<>k<>n,1<>i<>n;
    [a*(4*k-1) --> #]'f{i} : 1<>k<>n,1<>i<>n;
    [a*m --> a*2]'cl;
    [a --> a]'out "(a{2})+";
    [a --> a]'i' : 1<>i<>2;
    [a*2 --> #]'2;
    [a --> a]'3 : 2*n-1;
}

// Encoding module neuron_division_rules()

def neuron_division_rules(n)
{
    []'0 --> []'t{1} || []'f{1} "a";
    []'t{1} --> []'t{1+1} || []'f{1+1} "a" : 1<>i<>n-1;
    []'f{1} --> []'t{1+1} || []'f{1+1} "a" : 1<>i<>n-1;
}

// Encoding module neuron_budding_rules()
def neuron_budding_rules(n)
{
    [t’n] --> [t’n] / [cl "a";
    [f’n] --> [f’n] / [cl "a";
}

5.3 Simulation results

The pLinguaCore Java library includes a command-line interface in order to parse P-Lingua input files and simulate the defined P systems. The SN P system defined above can be simulated by writing the next command\(^1\) in a system console:

```
java -jar plinguacore.jar plingua_sim -pli sat_SNSystem.pli -o output.txt
```

A complete explanation of commands for pLinguaCore can be found in [7]. In this case, the P-Lingua input file is called sat_SNSystem.pli and it contains the source code referenced before. The file output.txt is a text file where information about the parser process and the generated computation is stored:

1. Initial cells
2. Initial multisets
3. Rules set
4. For each configuration:
   (a) Multiset of objects in the environment
   (b) Multiset of objects for each cell
   (c) Rules selected to be executed in the next step

The simulator runs until reaching a halting configuration, where no rule can be selected to be executed in the next step. At this stage, the environment contains either a single object \(a\) or no object at all.

For the P system defined in the file sat_SNSystem.pli, an object \(a\) is sent to the environment after 46 steps of computation and it halts. That is, it is an accepting computation.

6 Performance

In order to exemplify the simulation algorithm performance, a few execution examples are presented. All of them are referred to the family of SN P System solving SAT shown above, concretely 3-SAT instances. The execution environment is a Dual Core AMD Athlon II X2 250 3 GHz Speed 3 GB RAM computer running Windows XP Service Pack 3.

\(^1\) A Java runtime environment 1.6.0 or better must be installed. It can be downloaded from [http://www.java.com](http://www.java.com)
Table 1. Performance results

<table>
<thead>
<tr>
<th>variables (n)</th>
<th>clauses (m)</th>
<th>execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>0.734 s.</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.938 s.</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0.969 s.</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.578 s.</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1.937 s.</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4.016 s.</td>
</tr>
</tbody>
</table>

7 Conclusions and Future Work

In this paper we present a new release of P-Lingua, that significantly extends the previous version by incorporating the ability to work with Spiking Neural P systems (SN P systems). Besides, a new simulation algorithm has been designed and implemented, taking into account features of SN P systems such as neuron division and budding. This new simulator has been included into the library pLinguaCore, and it has been checked by simulating a family of SN P systems taken from the literature, for solving the well-known NP-complete problem SAT. A brief description of the solution to this problem using SN P systems has been included in the paper, along with the corresponding P-Lingua source code.

A possible course of future work is to include weights and thresholds, as described in Wang et al. [19], in P-Lingua and pLinguaCore simulator for SN P systems. The referred paper propose using weighted synapses, potentials in neurons, and rules which handle these potentials under the control of given firing thresholds.

In addition, it could be interesting to adapt MeCoSim (Membrane Computing Simulator, see [16]) for the simulation of SN P systems, making use of the new release of pLinguaCore. This would provide the end user of the new simulator a way to configure possible input and output tables to generate P-Lingua parameters and visualize the results of the computation in a graphical and structured way.

Finally, we are working along with other research groups to join forces to combine the expressive richness and flexibility of P-Lingua and MeCoSim with the efficiency of parallel simulators based on CUDA [1].

Acknowledgements

The authors acknowledge the support of the project TIN2009–13192 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds, and
the support of the Project of Excellence with Investigador de Reconocida Valía of the Junta de Andalucía, grant P08-TIC-04200.

References

17. G. Rozenberg. DNA processing in ciliates. The wonders of DNA computing in vivo. 
Unconventional models of Computation (UMC’2K), (2010), 116–118.
Division. Proceedings of the Eleventh International Conference on Membrane 
weights and thresholds. Proceedings of Tenth Workshop on Membrane Computing, 
2009, 514-533.
20. Java’s regular expressions specification:
http://download.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html
21. The Sun’s Java Web Site:
22. The P–Lingua Web Site:
http://www.p-lingua.org/
23. The SATLIB library:
http://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html
Computing with Multi-Membranes

Vincenzo Manca and Rosario Lombardo

Department of Computer Science, University of Verona
Strada Le Grazie, 15 – 37134 Verona, Italy
{vincenzo.manca, rosario.lombardo}@univr.it

Abstract. Multi-Membranes are introduced for defining a computation model inspired to Metabolic P systems. It is a deterministic, distributed, and computationally universal model, where computations are performed by transferring objects among membranes with fluxes specified by membrane contents. Arithmetical functions can be naturally described and, remarkably, the algorithms can be described by means of pure geometrical forms without any textual information.

Keywords: Membrane Computing, P Systems, Metabolic P Systems, Computation Models.

1 Introduction

Since the P systems were introduced [11], several variants of the model were defined and many software application have been used. P Systems can nondeterministically evolve to obtain different successful computations and are thus very suited in generating or recognizing languages, depending on the way the output is defined [11]. Membrane-like architectures with compartments communicating across membranes [10] and tissue-like P systems with some kind of graph structures [2, 1] were also considered in many computational and applicative contexts. Metabolic P systems, based on Gh. Păun’s membrane computing theoretical framework [12], were introduced to represent metabolic processes in a discrete mathematical setting. MP systems [5, 6] are deterministic single-membrane dynamical systems where multiset rewriting rules are regulated by functions. The encouraging results obtained by applying MP systems to the modeling of biological phenomena and to function approximation [7] suggested us the idea of defining a model of computation which is deterministic and based on rules regulated by fluxes.

In this paper, rather than generating or recognizing languages, we are interested in computing arithmetical functions using a novel computation model where calculi are essentially carried out by matter transfers. This intuition can be developed by considering a special kind of MP systems, where a new type of membrane structure is added in order to provide mechanisms for combining computations articulated at different distributed levels. The present paper continues an investigation started in [4], where the MP approach was developed in terms of substance transformations rather than matter exchange among membranes.

347
For the following discussion it is useful to briefly recall the conceptual framework of Metabolic P systems [6]. MP Systems are single-membrane dynamical systems. The rules are interpreted as reactions specifying variations of reactants and products. A multiset rule such as $2a + b \rightarrow c$ means that a number $2 \times n$ of molecules of kind $a$ and a number $n$ of molecules $b$ are replaced by $n$ molecules of type $c$. The value of $n$ is the flux of the rule application and is provided by a function called regulator of the reaction. Regulators take as arguments the variables in the system, that is, substance quantities (or physical parameters such as temperature, pressure) and provides the fluxes of reactions. The dynamics of an MP system is deterministic at population level and, at each step, is governed by a partition of matter determined by the fluxes of the rules consuming it.

**Example 1.** (MP grammars and MP graphs) An MP system is specified entirely by an MP grammar, where reactions are given with the corresponding regulators. The empty multiset $\emptyset$ on the left side (resp. right side) of the rule is used to specify introduction (resp. expulsion) of matter. An MP graph (see Fig. 1) is associated to the system, where the relationships between all substances, reactions and regulators appear in a more direct way.

$$r_1 : \emptyset \rightarrow x \quad \varphi_1 = 2 \quad (\varphi_1 \text{ is constant})$$
$$r_2 : x \rightarrow y \quad \varphi_2 = 2 \times y + x \quad (\varphi_2 \text{ depends on populations } x \text{ and } y)$$
$$r_3 : y \rightarrow \emptyset \quad \varphi_3 = 1 \quad (\varphi_3 \text{ is constant})$$

**Fig. 1.** Stoichiometric and regulatory components of an MP graph: substances (circles), input and output gates (tipping triangles on rules) are connected by reactions arrows. The regulators (rectangles) determining the flux of rules are linked with dashed arrows.

2 Multi-Membranes

The notion of membrane in P systems, according to Păun’s original idea, is based on the notion of structures which contain objects and possibly other membranes. In this sense, bracket notation is the natural way of expressing a configuration of membranes with their relative inclusions and the objects they contain.

The notion of membrane entails some topological aspects which are relevant for developing the extension that we will investigate in this paper. A closed
surface is a 2D compact variety without one-dimensional frontiers (see [8, 3] for more details and formal definition of the topological notions involved). Figure 2 shows some examples having different topological types of surfaces: a sphere, which does not have any 1D borders, and a torus, with any number of holes, are 2D compact surfaces (for avoiding complications, we exclude surfaces with knots, such as Klein bottles).

Fig. 2. Surfaces of increasing topological complexity. The left-most is the membrane concept introduced by Gh. Păun [12].

On the basis of the topological notion of a closed compact surface \( C \) in a 3D space \( S \) (the whole 3D space or the subspace bounded by another 3D surface including \( C \)), \( C \) identifies a membrane as a partition of \( S \) in three sets: i) the points of \( S \) on the surface of \( C \), which we call border of the membrane, ii) the points of \( S \) internal to the \( C \) iii) the points of \( S \) external to the \( C \).

Now we present a more complex notion of membrane where we add to the usual notion of inclusion among membranes also a junction relation among membranes which holds when two membranes share a portion of their borders (they are joined). An elementary multi-membrane is obtained by joining many elementary membranes. Starting from elementary multi-membranes, by iterating membrane inclusion and junction, general multi-membranes can be obtained (see Fig. 4). An elementary multi-membrane has a central structure if there is one central elementary membrane, to which are joined two or more elementary membranes, called the frontier membranes. A general multi-membrane with central structure is obtained, starting from an elementary multi-membrane with central structure, by means of hierarchical inclusions, into the central components, of membranes or multi-membranes having central structure (see Fig. 3).

The subspace of points internal to a membrane constitute its region. Any point of the space which is not on the border of a membrane belongs to the region of some membrane or is external to the region of any membrane, and in this case we say that it belongs to the most external region. By iterating inclusions and junctions it is possible to provide very complex spatial structures. In the following, for simplicity’s sake, we use 2D diagrams for representing multi-membranes, but we remark that membranes are intended as 2D forms in 3D spaces.
In the rest of the paper, we will show that multi-membranes with central structure can be used for defining a notion of deterministic computation, which is essentially based on the transfer of objects among membranes. However, the interest in investigating more complex notions of membranes has a general motivation, which is related to the way space is organized in biological systems. For example, endoplasmic reticulum and Golgi apparatus are examples of complex membrane structures which are essential for cell functions. Analogously, tissue and organ development, and morphogenesis phenomena imply forms of membrane organization, specialization, and conformation which cover an enormous spectrum of possibilities, which are crucial in the life strategies.

A further level of complexity can be introduced by channels. For defining channels we need to specify the notion of reachability. A membrane is reachable from a region if a portion of its border confines with the region. In particular, any membrane is reachable from its own region. In a region from which two membranes are reachable, a channel can be put between them, which goes from one of them, the source, to the other, the target. Along a channel, objects contained in
the source membrane can be transferred to the target membrane. The important aspect of a multi-membrane with central structure is the role of the frontier membranes which are joined to the central one. In fact they are reachable from the region of the central membrane but also from the points which are external to the central membrane and to other frontier membranes (see Fig. 5).

![Diagram of a multi-membrane with channels, representing transfer rules.](image)

**Fig. 5.** A multi-membrane with channels, representing transfer rules.

In a multi-membrane, a (transfer) rule is a channel between two membranes, both reachable from the region where the channel is located, which is equipped by a flux determining the quantity of objects passing from the source to the target membrane (in a given time unit). We denote a rule in the following way, where $a, b$ are membrane labels and $\varphi$ is the **flux**, that is, an expression assuming a value in correspondence to the values of its variables:

$$a \rightarrow b \# \varphi.$$  \hspace{1cm} (1)

The effect of such a rule is the passage of $\varphi$ objects from the membrane with label $a$ to the membrane with label $b$. In the specific cases we consider in this paper, $\varphi$ will be either a natural number or the label of some membrane. In the latter case the value of the flux is the number of objects inside the region of that membrane. In this sense, a rule can be seen as a **valve** which is modulated by membrane contents. The case of a flux specified by a natural number $k$ is equivalently represented by the label of a membrane containing $k$ objects where no rules is acting on it, so that its content remains unchanged in time. In this context we may assume that only one kind of objects is present in membranes, for example, if the entities are photons and channels photon fibers, a multi-membrane becomes a (futuristic) circuit acting on photon populations.

Introduction and expulsion rules are a special case of the rules considered above, where the source or the target is the most external region (denoted also by the empty membrane $\emptyset$). The effect of rules such as $\emptyset \rightarrow b \# \varphi$ or $a \rightarrow \emptyset \# \varphi$ are the introduction (entering from the external environment) or expulsion (exiting to the external environment) of $\varphi$ objects respectively. The intuition behind such rules, also for membranes which do not confine with the most external region (when they are placed at a deeper inclusion level) could be motivated in
many possible ways. A possible implementation of this mechanism is given in Fig. 6 by means of cascades of buffer frontier membranes providing input and output communication channels with the environment of the multi-membrane.

3 Computing with Multi-Membranes

The deterministic computation model based on multi-membranes is based on the following definition, for which examples are provided further on in this section.

**Definition 1.** (Multi-Membrane Computation System) A Multi-Membrane System with central structure is a construct:

\[ M = (\mathcal{L}, R, \mu) \]

where:

- **\( \mathcal{L} \)** is a set membrane labels.
- **\( R \)** is a set of multi-membrane rules having the following forms with \( a, b \in \mathcal{L} \) and \( \varphi \in \mathbb{N} \) or \( \varphi \in \mathcal{L} \):
  - trans: \( a \rightarrow b \# \varphi \)
  - extra-in: \( \emptyset \rightarrow b \# \varphi \)
  - extra-out: \( a \rightarrow \emptyset \# \varphi \)
- **\( \mu \)** is the initial multi-membrane configuration with only one type of objects, that is, a multi-membrane with central structure where some initial objects are inside certain membranes (by default assumed empty).

In each multi-membrane some frontier membranes are marked by:

- **START** one and only one membrane is marked by START;
- **HALT** one and only one is marked by HALT;
- **IN, OUT** some membranes are marked by IN and some with OUT, but these marks may not occur.

---

**Fig. 6.** A multi-membrane with 3 rules where the flux of \( r_1 \) is defined as the contents of membrane \( m_2 \). Rounded rectangles represent possible buffer frontier membranes.
Rules in the system are applied according to the following general principle. The rules sharing a common source membrane are simultaneously applied if all their fluxes are defined and if the objects that they globally extract from the source are less than the number of objects inside the source membrane. If this is not the case no rule can be applied. Moreover, a flux given by the content of a frontier membrane is defined only when the multi-membrane of this frontier membrane has one object in its \textit{Halt} membrane.

Multi-membrane systems compute functions according to the following Def. 2.

\textbf{Definition 2.} (MM-computable function) A function \( f : \mathbb{N}^k \rightarrow \mathbb{N}^h \) such that \( f(x_1, \ldots, x_k) = (y_1, \ldots, y_h) \) is MM-computable if there exist a multi-membrane System \( M \) with central structure where:

- Frontier membranes marked by \textit{In} contain the values \( x_1, \ldots, x_k \) respectively,
- the contents of frontier membranes \textit{Start} and \textit{Halt} are 0 and the system is setup with its initial configuration,
- provided that \textit{Start} is only referred as target from outer region and only as source from inner region (and reversed for \textit{Halt}), after posing one object in the membrane marked \textit{Start} (as indication of “computation in progress”) the rules of the system get applied according to Def. 1. The system \( M \) eventually ends when concomitantly \textit{Start} is emptied and \textit{Halt} receives one object (as indication of “end of computation”). In this case, the values \( y_1, \ldots, y_h \) which are in the frontier membranes marked by \textit{Out} provide the results of the computation, and all internal rules become inapplicable.
- Moreover, the \textit{Halt} membrane never contains an object iff \( f(x_1, \ldots, x_k) \) is undefined.

3.1 Examples of Multi-Membrane Systems

Multi-membrane systems are represented with three equivalent formalisms. In \textit{Annotated graphs}, the graphical elements of Table 1 are accompanied by textual annotations defining membrane labels and flux definitions (see Fig. 7).

\textit{Bracketed notation} provide a complete textual specification combining the multi-compartmental configuration \( \mu \) with the rules, which could constitute the basis for a textual multi-membrane programming language. The passage from the annotated graphs to the bracketed notation is explained in Fig. 7, where the at-symbol \( @ \) indicates that the following labelled membranes (of types \textit{Start}, \textit{Halt}, \textit{In}, \textit{Out}) are frontier membranes.

\textit{Pure multi-membrane graphs} completely express multi-membrane algorithms using a pure visual formalism. In this case, a flux referring a membrane is denoted by dotted lines (eg. \( \varphi = m_2 \) in Fig. 6), and any constant flux (eg. \( \varphi = 3 \)) is replaced by a reference to an isolated membrane containing the same number of objects (eg. \( \varphi = t \) and \( [t, 3]_t \)).

Sometimes we use a summation in the flux, such as \( r_1: a \rightarrow b ~ \# \varphi + \psi \), meaning that a multiset is removed from \( a \) and placed into \( b \) with a flux \( \varphi + \psi \), actually being the result of two rules whose fluxes are \( \varphi \) and \( \psi \) respectively.
Table 1. Graphical elements denoting the frontier membranes in a multi-membrane computation system. In annotated graphs (see Fig. 7) textual annotations are placed besides membrane labels and the flux definitions.

<table>
<thead>
<tr>
<th>Multi-membrane system</th>
<th>Graphical symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>⏞</td>
<td>In</td>
</tr>
<tr>
<td>Out</td>
<td>♂</td>
<td>Out</td>
</tr>
<tr>
<td>In-Out</td>
<td>⨯</td>
<td>In-Out</td>
</tr>
<tr>
<td>START</td>
<td>⬤</td>
<td>START</td>
</tr>
<tr>
<td>HALT</td>
<td>⬤</td>
<td>HALT</td>
</tr>
<tr>
<td>⬤⬤</td>
<td></td>
<td>Rule $a \rightarrow b \ # \varphi$</td>
</tr>
<tr>
<td>⬤⬤</td>
<td></td>
<td>Expulsion rule $a \rightarrow \emptyset \ # \varphi$</td>
</tr>
<tr>
<td>⬤⬤</td>
<td></td>
<td>Introduction rule $\emptyset \rightarrow b \ # \varphi$</td>
</tr>
</tbody>
</table>

Fig. 7. Bracketed notation combines the initial configuration $\mu$ with the rules of the system. Frontier membranes $x$ and $y$ (prefixed by the $@$ symbol in bracket notation) initially contain 3 and 2 objects respectively, while rules $r_1$ and $r_2$ are localized in membrane SUB.
Limited Subtraction. The limited subtraction, written as $x \odot y$, is properly defined when the subtrahend can actually be subtracted from the subtracter. More formally:

$$\odot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, \quad x \odot y = \begin{cases} x - y & \text{if } x \geq y \\ x & \text{otherwise} \end{cases}.$$  

The annotated graph in Fig. 8 shows a multi-membrane system implementing the operation in one single step. When $\text{Start}[x]_s$ becomes 1, the computation begins and rule $r_1$ tries to expel matter from $x$ with a flux equal to the subtrahend $y$. If there is not enough matter $r_1$ blocks and the input is left untouched. Otherwise in membrane $[x]_x$ remains the value $x - y$ of a proper subtraction. In the same step the computation is halted by rule $r_2$ with a flux $s = 1$.

After the system has provided the result, the input membranes could need a set-up procedure before another computation can be started correctly.

Fig. 8. Annotated graph for the limited subtraction. The frontier membrane $x$ acts both as input and output of the computation.

The multi-membrane system is specified in bracketed notation as:

$$[\text{Sub} \odot : \text{Start}[x]_s \quad \text{Halt}[h]_h \quad \text{InOut}[x]_x \quad \text{In}[y]_y \quad ; \quad r_1 : x \to \emptyset \# y \quad r_2 : s \to h \# s]_{\text{Sub}}$$

Multiplication. An algorithm for performing the multiplication of two numbers $a$ and $b$ is the repeated addition of factors. The multi-membrane in Fig. 9 implements $a \times b$ as the sum $b + \cdots + b$.

The computation starts when $s = 1$. Suppose $a > 0$, $b \geq 0$. At step 1, $a$ is decreased by 1 unit, while the rule $r_2$ is blocked $(b + a > b)$, i.e. the value $b$ does not change. In the same step $r_3$ introduces $b$ units into $c$ while other rules are blocked. The same dynamics is repeated at following steps while $b + a > b$ or, equivalently, while $a > 0$. At step $i = a$ the contents of $a$ finally passes from 1 to zero, while the contents of $b$ does not change.
Fig. 9. Multiplication network for $a \times b$ is based on repeated addition.

At next step $r_3$ adds additional $b$ units to $c$ (i.e. there are $(a + 1) \times b$ units in membrane $c$) while $r_2$ now has a flux $\#b$ and can move the contents of $b$ into $b'$.

Therefore in next step $a + 2$ we have: i) $a = 0$ and $b = 0$ ii) $r_3$ is blocked iii) $r_4$ expels $b' = b$ units from $c$ iv) $r_5$ halts the system. Frontier membrane $\text{OUT}_c$ now contains $a \times b$ units.

The same reasoning holds for both $a = 0$ or $b = 0$.

The multi-membrane system for the multiplication is defined by:

\[
\left[ \text{Mul} \right] = \left[ \text{START}_a \right] \text{Halt}_b \text{IN}_c \text{IN}_d \text{OUT}_e ;
\]

\[
\begin{align*}
    r_1 : a & \rightarrow \emptyset \#1 \\
    r_2 : b & \rightarrow b' \#a + b \\
    r_3 : \emptyset & \rightarrow c \#b \\
    r_4 : c & \rightarrow \emptyset \#b' \\
    r_5 : s & \rightarrow h \#1 + a + b
\end{align*}
\]

The pure graph for the multiplication (see Fig. 10) is a topological form which entirely describes an algorithm in terms of a pure visual formalism. Fluxes that were previously defined using the constant 1, here are modeled by references to an isolated membrane containing 1 object.

3.2 Computational Universality

The multi-membrane systems are a universal computation model as stated in the following proposition.

**Proposition 1.** Multi-membrane systems have the same computational power of register machines.

Before proving the proposition, we preliminarily introduce a multi-membrane system describing the conditional construct useful in defining the conditional jumps in a register machine.
The pure graph for the multiplication of two input numbers contains no textual annotation and numerical constants are modeled by isolated membranes.

**Conditional Construct.** The conditional construct (see Fig. 11) is intended to be reusable in possible loops performed by the register machine on the instruction sequence. The idea is to place a single object in one of two alternative membranes depending on the value of a guard membrane \( z \). We are defining the function:

\[
\text{Cond}(z) = \begin{cases} 
(1, 0) & \text{if } z = 0 \\
(0, 1) & \text{if } z = 1.
\end{cases}
\]

The multi-membrane system for the conditional construct is described by:

\[
\begin{align*}
\text{Cond} & : \text{Start}_s \quad \text{Halt}_h \quad \text{In}_z \quad \text{Out}_x \quad \text{Out}_y \\
& : s \rightarrow t \quad # 1 \\
r_1 : h \rightarrow x \quad # s \\
r_2 : x \rightarrow y \quad # z \\
r_3 : t \rightarrow h \quad # 1
\end{align*}
\]
When $\text{START}_s$ becomes 1, the computation begins with $r_1$ moving 1 object from $s$ to $t$ while rule $r_2$ introduces 1 object into membrane $x$. According to the multi-membrane Def. 2 no matter can be extracted from outputs $x$ or $y$ until $\text{HALT}_h$ has been produced. At step 2 we consider two cases.

Case i) input $z = 0$: while rule $r_3 : x \to y \# z$ is blocked, $r_4 : t \to h \# 1$ signals the end of computation and the output is $(x, y) = (1, 0)$.

Case ii) input $z = 1$: rule $r_3 : x \to y \# z$ acts by inverting the boolean values $x$ and $y$, leading to the output $(x, y) = (0, 1)$.

Proof. of Proposition 1. We consider a definition of register machine which is a variant of the Minsky’s model given in [9], having the following types of instructions:

- Increment of register $R$, denoted with $\text{INC}(R)$.
- Decrement of register $R$, denoted with $\text{DEC}(R)$.
- Go-to instruction $I_k$ if register $R_j \neq 0$, denoted with $\text{JNZ}(R_j, I_k)$.
- HALT, stopping the computation.

For example the sum of two numbers greater than zero, which are put in registers $R_1$ and $R_2$, is given by the contents of register $R_1$ at the end of the computation of the following program:

1 : $\text{INC}(R_1)$
2 : $\text{DEC}(R_2)$
3 : $\text{JNZ}(R_2, 1)$
4 : HALT

In general a register machine is defined by a set $R_1, \ldots, R_n$ of registers and a program which is a sequence of labelled instructions $I_1, \ldots, I_m$ of the types given above, where labels identify the positions of the instructions in the program. A computation of this machine is obtained by putting some numbers in some specified input registers and by executing the instructions of the program in the order they are (where a go-to instruction specifies as next instruction to execute one which possibly is not the following one in the sequential order). When the HALT instruction is executed, the results of the computation are the numbers put in some output registers.

We construct a multi-membrane system modeling a register machine as follows (see also Fig. 13). We use INOUT frontier membranes $R_1, \ldots, R_n$ for each register in the machine and we provide simple inner membranes $I_1, \ldots, I_m$ modeling the labels of instructions. We add rules $s \to I_1 \# 1$ and $I_m \to h \# 1$ connecting $\text{START}_s$ and $\text{HALT}_h$ with the first and last instruction-membrane respectively. One single control object is supposed to flow from $\text{START}_s$ to $\text{HALT}_h$ through the instruction-membranes $I_1, \ldots, I_m$.

For any $\ell, 1 \leq \ell \leq m$, if instruction $I_\ell$ is an increment instruction then we place an incrementing rule $\emptyset \to R \# I_\ell$, while if $I_\ell$ is a decrement instruction then we place a decrementing rule $R \to \emptyset \# I_\ell$. Since a register can be referenced
Fig. 12. Four instructions $I_1, I_2, I_3, I_4$ acting on two register $R_1, R_2$. $I_1$ and $I_3$ are increment instructions acting upon $R_1$, while $I_2$ and $I_4$ act upon $R_2$: the former decrementing and the latter incrementing register $R_2$.

by several INC or DEC instructions, there will be several input/output rules acting (at different times) on the corresponding frontier membrane (see Fig. 12).

We denote with $R^+$ and $R^-$ the set of labels of instructions incrementing and decrementing, respectively, register $R$. According to the sequential semantic of the register machine, there is always one and only one instruction which is executing at any given step.

If the instruction $I_\ell$ is a $\text{Jnz}(R_j, I_k)$, we properly embed the conditional construct (the striped multi-membrane in Fig. 13) into the network.

Fig. 13. The multi-membrane system modeling a register machine.

Let $[I_\ell]_1$ be the START of the conditional construct. If $R_j = 0$, then $\text{IN}[z]_1$ stays empty and, according to the definition of $\text{COND}$, the output is returned
as \text{OUT}[x, 1]_x \text{ which is moved as control object into } I_{\ell+1}. \text{ Otherwise if } R_j > 0, \text{ membrane } I_n[z]_z \text{ is filled by a rule whose flux is } 1 = \#I \text{ units. Afterwards, while a rule restores the original value from } z \text{ into } R_j, \text{ the control object is passed from membrane } y \text{ to } R_k.

\[\square\]

3.3 More Complex Examples of Computation

\textbf{Abacus.} In this example is defined a 3-digit abacus with carry in base 10. Two input numbers \( N = 728 \) and \( M = 322 \) are encoded inside the input membranes as \([N_1, 7]_{N_1}, [N_2, 2]_{N_2}, [N_3, 8]_{N_3}\) and \([M_1, 3]_{M_1}, [M_2, 2]_{M_2}, [M_3, 2]_{M_3}\) (see Fig. 14). The objects from input membranes are summed digit-by-digit into the output membranes. The membrane \text{OUT}[R]_R \text{ encodes the digit of thousands possibly yielded by the carry rules which try to consume from their neighboring lower-digit membrane in order to produce a carry unit into their neighboring higher-digit membrane. The computation, with the carry operations, may take up to 4 steps counted by an auto-resetting countdown that sets } h = 1 \text{ and confirms the final output.}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig14}
\caption{A 3-digit abacus with carry.}
\end{figure}

When \text{START}[s]_s \text{ becomes 1, the contents of } t \text{ is transferred into } k. \text{ In the following steps } 2 \text{ and } 3, \text{ rule } r_{10} : k \rightarrow t \#1 \text{ decrements down to } 1 \text{ the content of membrane } [k]_k. \text{ At step } 4 \text{ the contents of membrane } [k]_k \text{ is } 1 \text{ and the rule } r_{13} : s \rightarrow h \#k \text{ can be applied halting the computation.}

The overall efficiency, as in digital computers, has an important limitation in the carry mechanism unless some additional optimizations are implemented.
The system is specified in bracketed notation where the rules without a name are said to be anonymous (eg. the helper rules for the carry extraction):

\[
\text{Abacus}
\]

\[
\begin{align*}
\text{Init} & : \text{START}_s \text{halt} \ ; \\
\text{In} & : [N_1, 7][N_1, 2][N_2, 8][N_3, 3][M_1, 2][M_2, 2][M_3, 2] \\
\text{Out} & : [R, R_1, R_2, R_3] \\
\text{k} & : \text{t} \ ; \\
\text{r}_1 & : N_1 \rightarrow R_1 \ # \text{N}_1 \\
\text{r}_2 & : M_1 \rightarrow R_1 \ # \text{M}_1 \\
\text{r}_3 & : N_2 \rightarrow R_2 \ # \text{N}_2 \\
\text{r}_4 & : M_2 \rightarrow R_2 \ # \text{M}_2 \\
\text{r}_5 & : N_3 \rightarrow R_3 \ # \text{N}_3 \\
\text{r}_6 & : M_3 \rightarrow R_3 \ # \text{M}_3 \\
\text{r}_7 & : R_3 \rightarrow R_2 \ # \text{1} \\
\text{r}_8 & : R_2 \rightarrow R_1 \ # \text{1} \\
\text{r}_9 & : R_1 \rightarrow R \ # \text{1} \\
\text{r}_{10} & : k \rightarrow t \ # \text{1} \\
\text{r}_{11} & : t \rightarrow k \ # \text{3} \\
\text{r}_{12} & : s \rightarrow h \ # \text{k}
\end{align*}
\]

**Division.** The division of two natural numbers, written \(a \div b\), is a function

\[a \div b = (q, r)\text{ where }a = q \cdot b + r, \ 0 \leq r < b\ .\]

The idea behind the implementation of \(a \div b\) is based on an iterative subtraction of the divisor \(b\) from the dividend \(a\). The remainder of such subtractions, if any, is left into the dividend membrane \(a\).

To keep track of the number of repeated subtractions from \(a\) (i.e. the quotient) we remove separately the numbers \(b - 1\) and \(1\). The number of \(1s\) that are put into membrane \(Q\), gives the quotient (see Fig. 15), while the predecessor \(b - 1\) is computed by inner multi-membrane \([\text{pred}]\text{pred}\). Rules \(r_4\) and \(r_5\) perform the repeated subtractions with fluxes depending on the result \([\text{OUT}]\text{OUT}\) computed by multi-membrane \([\text{pred}]\text{pred}\). According to Def. 1, they can be applied both or none guaranteeing that exactly \((b - 1) + 1 = b\) units are subtracted. Additional logic is included to determine the end of computation (signaled by the content of membrane \([f]f\)) and restoration of initial input \([\text{IN}]\text{IN}\) (as sum of the contents of \([\text{OUT}]\text{OUT}\) and \([f]f\)).

The annotated graph for \(\text{Div}\) in Fig. 15 is equivalently expressed in bracketed notation:

\[
\text{Abacus}
\]

\[
\begin{align*}
\text{Div}
\end{align*}
\]
Proposition 2. The multi-membrane system $\text{Div}$ computes the function $\delta$ such that:

$$\forall a, b \in \mathbb{N}, \quad \delta(a, b) = (q, r) \quad \text{where} \quad \{ q \text{ is the quotient of } a \div b \}$$

$$\{ r \text{ is the reminder of } a \div b \}$$

Fig. 15. Division $a \div b$ is based on repeated subtractions.
Proof. If we have \( a > b > 0 \), there is an initialization phase of two steps. At step 1 two copies of \( b \) are stored into \( b' \) and \( b'' \), while \( r_3 \) gives the start to multi-membrane \( \text{[pred]} \). At step 2, multi-membrane \( \text{[pred]} \) produces the \( \text{OUT}[b', b - 1]_b \) and \( \text{HALT}[b, 1]_b \) which in next steps will unblock the rules \( r_4 \) and \( r_5 \) for the repeated subtractions.

If \( q \) and \( r \) are the quotient and reminder of \( a \div b \), we know that the invariant \( a - ib = (q - i)b + r \) holds at each step of subtraction \( i \), \( i = 0, \ldots, q \). Therefore, after the initialization, we consider the invariant at steps \( 2 + i \), \( i = 0, \ldots, q \) for the two cases whether the division has a reminder or not.

Case \( i) \) \( r > 0 \): for input \( (a, b) \) with \( a > b > 0 \) and \( a \div b \) yielding \( a = q \cdot b + r \). After \( q \) subtractions, membrane \( Q \) contains exactly \( q \) units, with \( q \geq 1 \), and the invariant above assumes the form \( a - q \cdot b = r \) meaning that membrane \( a \) now contains \( r \) units, with \( r < b \). There are now three finalization steps. At step \( 2 + q + 1 \) rule \( r_6 \) can be applied. At step \( 2 + q + 2 \) rule \( r_7 \) moves 1 unit from \( [n]_a \) into \( [f]_f \). Finally rules \( r_8, r_9 \) and \( r_{10} \) are fired: the former two restore the original content of \( [b]_b \) and the latter halts the computation.

Case \( ii) \) \( r = 0 \): for input \( (a, b) \) with \( a \geq b > 0 \) and \( a \div b \) yielding \( a = q \cdot b \).

After \( q - 1 \), subtractions in membrane \( a \) are left exactly \( b \) units and rule \( r_6 \) is applied. At next step \( 2 + q \), \( r_4 \) and \( r_5 \) complete the division while \( r_7 \) triggers the termination for subsequent step \( 2 + q + 1 \).

If we have \( b = 0 \), then at step 1 all rules are blocked (except \( r_3 \)). In this case the system never puts an object into \( \text{HALT}[\text{[}] \) because a division by zero is undefined. However it is possible to implement an extension of the division which always halts and returns an additional output set to 1 in case \( b = 0 \). More cases from Table 2 can be easily verified on the annotated graph in Fig 15. \( \Box \)

<table>
<thead>
<tr>
<th>Initial condition</th>
<th>Reminder in ( a )</th>
<th>Quotient in ( Q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \geq b &gt; 0 ), ( a = q \cdot b )</td>
<td>0</td>
<td>( q )</td>
</tr>
<tr>
<td>( a &gt; b &gt; 0 ), ( a = q \cdot b + r )</td>
<td>( r )</td>
<td>( q )</td>
</tr>
<tr>
<td>( b = 0 ) (undefined)</td>
<td>( a )</td>
<td>0</td>
</tr>
<tr>
<td>( b &gt; a = 0^1 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b &gt; a &gt; 0 )</td>
<td>( a )</td>
<td>0</td>
</tr>
</tbody>
</table>

4 Conclusions

In this paper the original idea of MP-Systems was combined with a new notion of membrane, we call multi-membrane, where the junction of membranes is used...
together with the inclusion of membranes. In this membranes a new computa-
tion model is defined which seems to be appropriate for complex arithmetical
computations, by designing a sort of circuits in pure geometrical forms (by draw-
ing multi-compartment, transfer edges and regulating edges). Topics of future
research in this direction will be developed along two different lines: i) investigat-
ing on the computational expressiveness of multi-membrane systems by means
of more complex arithmetic functions (many operations in modular arithmetic
result to be naturally MM-computable), or by means of software tools extending
a previous tool developed in the context of [4], and ii) exploring the biological
meaning of multi-membranes, or suitable extensions, as structures relevant in
morphogenetic phenomena.

References

on Natural and Artificial Computation, LNCS 6686. J. Ferrández at al. (Eds.).
6. V. Manca. Fundamentals of Metabolic P Systems. The Oxford Handbook of Mem-
10. R. Nicolescu, M. Dinneen, and Y. Kim. Towards Structured Modelling with Hyper-
dag P Systems. International Journal of Computers, Communications and Control,
A methodology based on MP theory for gene expression analysis

Luca Marchetti and Vincenzo Manca
University of Verona, Department of Computer Science
Strada Le Grazie 15, 37134 Verona, Italy
luca.marchetti@univr.it, vincenzo.manca@univr.it

Abstract. In this paper we develop an application of the MP theory to gene expression analysis. After introducing some concepts about DNA microarray experiments and about gene networks, we delineate a methodology to model and simulate such kind of networks by means of Metabolic P systems. MP systems were initially introduced to model metabolic processes, but they can be successfully used in each context where we want to infer models of a system from a given set of time series. In the case of MP modelling for gene expression analysis, we found a standard way for translating MP grammars involving gene expressions into corresponding quantitative gene networks. Pre-processing methods of raw time series, coming from microarray analysis, have been also elaborated in order to permit a successful MP modelling of the underlying gene network.

1 Introduction

A remarkable development in molecular biology today is the upscaling to the genomic level of its experimental methods [16]. Hardly imaginable only 20 years ago, the sequencing of complete genomes has become a routine job, highly automated and executed in a quasi-industrial environment. The miniaturization of techniques for the hybridization of labelled nucleic acids in solution to DNA molecules attached to a surface has given rise to DNA microarrays, tools for measuring the level of gene expression in a massively parallel way [21].

The basic design principle is the same for all forms of microarray, whether based on DNA, proteins or cells. Specific molecular targets are detected simultaneously within the sample of interest by an array of probes. The probes, often numbering tens of thousands, are chemically attached in an array format to a solid substrate to construct the microarray (see Figure 1). The significance of the microarray to the field of molecular biology is the parallel detection capabilities of the system. Microarrays offer the ability to achieve simultaneous detection of many targets, and through optimization this can be achieved without detriment to sensitivity [20].

One of the most important applications for arrays so far is the monitoring of gene expression (mRNA abundance). Unfortunately, the billions of bases of DNA sequence do not tell us what all the genes do, how cells work, how cells
form organisms, what goes wrong in disease, how we age or how to develop a drug. This is where functional genomics comes into play. The goal is not simply to provide a catalogue of all the genes and information about their functions, but to understand how the components work together to comprise functioning cells and organisms.

The collection of genes that are expressed or transcribed from genomic DNA, sometimes referred to as the expression profile or the transcriptome, is a major determinant of cellular phenotype and function. The transcription of genomic DNA to produce mRNA is the first step in the process of protein synthesis, and differences in gene expression are responsible for both morphological and phenotypic differences as well as indicative of cellular responses to environmental stimuli and perturbations. Unlike the genome, the transcriptome is highly dynamic and changes rapidly and dramatically in response to perturbations or even during normal cellular events such as DNA replication and cell division [21]. In terms of understanding the function of genes, knowing when, where and to what extent a gene is expressed is central to understanding the activity and biological roles of its encoded protein. In addition, changes in the multi-gene patterns of expression can provide clues about regulatory mechanisms and broader cellular functions and biochemical pathways. In the context of human health, the knowledge gained from these types of measurements can help determine the causes and consequences of a disease, how drugs and drug candidates work in cells and organisms, and what gene products might have therapeutic uses themselves or may be appropriate targets for therapeutic intervention.
Microarray analysis offers also a promising approach to understand what goes wrong in cancerous transformed cells. Causative effects and potential therapeutic targets can be identified by determining which genes are upregulated in different tumour types [36, 37], and specific candidate genes can be intentionally overexpressed in cell lines or cells treated with growth factors in order to identify downstream target genes and to explore signalling pathways [9, 11]. Tumorigenesis is often accompanied by changes in chromosomal DNA, such as genetic rearrangements, amplifications or losses of particular chromosomal loci, and developmental abnormalities, such as Down’s or Turner’s syndrome, may arise from aberrations in DNA copy number. Since genomic DNA can be interrogated in much the same way as mRNA, comparisons of the copy number of genomic regions or the genotype of genetic markers can be used to detect chromosomal regions and genes that are amplified or deleted in cancerous or pre-cancerous cells. By using arrays containing probes for a large number of genes or polymorphic markers, changes in DNA copy number have been detected in breast cancer cell lines and in other tumours [34, 33]. The identification of when and where changes in copy number or chromosomal rearrangements have occurred can be used in both the classification of cancer types and the identification of regions that may harbour tumour-suppressor genes.

In this paper, we will discuss methods to apply the MP theory for gene expression analysis by modelling and simulating a particular type of molecular interaction network, called gene network. Today, a large part of the experimental data available, notably gene transcription data, concerns these networks of genes, proteins, and their mutual interactions.

2 Gene networks

A very successful model in biochemistry is a depiction of relationships between molecules as networks of interactions. These biochemical networks can be constructed at several levels and can represent different types of interaction. Several biochemical networks have traditionally been considered: i) metabolic networks representing the chemical transformations between metabolites; ii) protein networks representing protein-to-protein interactions, such as the formation of complexes and protein modification by signaling enzymes (also known as signalling networks); and iii) gene networks representing relationships that can be established between genes, when observing how the expression level of each one affects the expression level of the others. Each of these types of network is a simplification of the complete cellular system, which we refer to as the global biochemical network to emphasize that it explicitly includes all three types of molecule (metabolites, proteins and mRNA). Adoption of those simplifications for description of specific phenomena depends largely on which cellular components were observed experimentally. Thus, when exclusively monitoring gene expression to study some phenomenon, one is limited to constructing a gene network to explain the data.
Fig. 2. An example of a global biochemical network [3]. Molecular constituents (nodes of the network) are organized in three levels (spaces): mRNAs, proteins, and metabolites. Solid arcs indicate interactions (arrows mean activation, bars mean repression). Three different mechanisms of gene-to-gene interactions are shown: i) regulation of Gene 2 by the protein product of the Gene 1; ii) regulation of the Gene 2 by the Complex 3-4 formed by the products of Gene 3 and Gene 4; and iii) regulation of Gene 4 by the Metabolite 2, which in turn is produced by Protein 2. Projections of these interactions into the “gene space”, indicated by dashed lines, constitute the corresponding gene network (see also Figure 3).

Figure 2 represents a model of a global biochemical network in which the three levels are shown explicitly as planes. In any global biochemical network, genes do not interact directly with other genes; instead, gene induction or repression occurs through the action of specific proteins, which are, in turn, products of certain genes. Gene expression can also be affected directly by metabolites, or through protein–metabolite complexes. However, it is often useful to abstract these actions of proteins and metabolites, and represent genes acting on other genes in a gene network (also called genetic regulatory, transcription or expression networks). This simplification of going from the global biochemical network to a gene network is akin to a projection of all interactions to the “gene space” (see Figure 3).

Traditional molecular biology might have propagated the idea that genes dictate all that goes on inside a cell. This materialized in the central “dogma” of molecular biology, which emphasizes that proteins, and consequently metabolites, are only synthesized when genes are activated. This “dogma” failed to acknowledge that gene expression is also influenced by the levels of protein and metabolite. Systems in which there is no feedback from proteins or metabolites to
genes are called *dictatorial*, but are currently only used conceptually. It is now well established that regulation is distributed over all levels, and accordingly such systems are referred to as *democratic* [40].

Although such gene networks are phenomenological because they do not explicitly represent the proteins and metabolites that mediate cell interactions, they are a logical way of describing phenomena observed with transcription profiling, such as those that occur with microarray technology. The ability to create gene networks from experimental data and to use them to reason about their dynamics and design principles will increase our understanding of cellular function [3, 7]. Gene networks provide a large-scale, coarse-grained view of the physiological state of an organism at the mRNA level. The mRNA phenotype can be a very important representation of cell function, offering much more precise description than the one which can be achieved with words, even when these words are part of a controlled vocabulary, such as the *Gene Ontology*TM [39].

Cells exhibit complex interacting behaviour that is usually not predictable from the properties of individual system components alone. Gene networks provide such a system view at the level of gene activities. The detailed molecular mechanisms of how the products of one gene affect expression of another gene are often unknown but the effect itself can be easily observed in gene-expression experiments. It is therefore appropriate and timely to use genome-wide gene-expression data to identify gene networks, an important step towards uncovering the complete biochemical networks of cells. Knowledge about gene networks might provide valuable clues and lead to new ideas for treating complex diseases. It will aid pharmaceutical research in prioritizing targets, tailoring drug therapy to the individual needs of each patient, and can form the basis for rational gene therapy.

### 3 MP modelling of gene networks

The identification of gene networks from microarray data are now an important part of systems biology [14, 18]. In addition to high-throughput experimental methods, mathematical and computational approaches are indispensable for the analysis of gene networks. Given the large number of components of most networks of biological interest, connected by positive and negative feedback loops,
an intuitive comprehension of the dynamics of the system is often difficult, if not impossible to obtain. Mathematical modelling supported by computer tools can contribute to the analysis of a regulatory network by allowing the biologist to focus on a restricted number of plausible hypotheses. Many reviews of the modelling and simulation of gene networks have been published in recent years (e.g. [2, 10, 15, 12, 38, 4]), presenting the wide variety of formalisms that have been proposed in the literature, such as oriented graphs, Bayesian networks, Boolean networks, differential equations, stochastic master equations and stochastic P systems. In this paper we will discuss a methodology in order to model and simulate gene networks by means of Metabolic P systems [26, 23, 24, 22].

Metabolic P systems (MP systems), based on Păun’s P systems [31], were introduced for modelling metabolic systems in contrast to the classical continuous approach based on ordinary differential equations (ODEs). In MP systems no instantaneous kinetics are addressed, but rather the variation of the whole system under investigation is considered, at discrete time points, separated by a specified macroscopic interval $\tau$. The dynamics is given along a sequence of steps and, at each step, it is governed by partitioning the matter among reactions which transform it. Metabolic P systems proved to be promising in many contexts and their applicability was tested in many situations where differential models are prohibitive due to the unavailability or the unreliability of the kinetic rates [22, 29, 28, 30, 5].

A Metabolic P system is essentially a multiset grammar where multiset transformations are regulated by functions. Namely, a rule like $a + b \rightarrow c$ means that a number $u$ of molecules of kind $a$ and $u$ of kind $b$ are replaced by $u$ molecules of type $c$. The value of $u$ is the flux of the rule application. Assume to consider a system at some time steps $i = 0, 1, 2, \ldots, t$, and consider a substance $x$ that is produced by rules $r_1, r_3$ and is consumed by rule $r_2$. If $u_1[i], u_2[i], u_3[i]$ are the fluxes of the rules $r_1, r_2, r_3$ respectively, in the passage from step $i$ to step $i + 1$, $i \in \mathbb{N}$, the set of natural numbers, then the variation of substance $x$ is given by:

$$x[i + 1] - x[i] = u_1[i] - u_2[i] + u_3[i].$$

In an MP system it is assumed that in any state the flux of each rule $r_j$ is provided by a state function $\varphi_j$, called regulator of the rule. We call tuners of a regulator the set of variables (substances and parameters) which occur in its corresponding formula. Substances, reactions, and regulators (plus parameters which are variables different from substances occurring as arguments of regulators) specify the following discrete dynamics ($x[i]|i \in \mathbb{N}$) for any substance $x$, starting from the given value $x[0]$, called EMA (Equational Metabolic Algorithm):

$$x[i + 1] = x[i] + \sum_{j=1}^{m} \alpha_j u_j[i]$$

where $m$ is the number of rules and $\alpha_j$ are the coefficients of fluxes acting on substance $x$.

MP systems are equipped with a powerful regression algorithm, called Log-Gain Stoichiometric Stepwise Regression (LGSS), which derives MP models from
the time series of observed dynamics and that can be applied independently from any knowledge about reaction rate kinetics [27]. LGSS represents the most recent solution, in terms of MP systems, of the inverse dynamics problem, that is, of the identification of (discrete) mathematical models exhibiting an observed dynamics and satisfying all the constraints required by the specific knowledge about the modelled phenomenon. The LGSS algorithm combines and extends the log-gain principles developed in the MP system theory [24, 22] with the classical method of Stepwise Regression [13], which is a statistical regression technique based on Least Squares Approximation and a statistical F-test [8]. The method can be correctly applied independently from any knowledge about reaction rate kinetics, and can provide, with respect to differential models, different and even simpler mathematical formulations [28, 30].

In terms of the notation introduced in Section 2 about global biochemical networks (see also Figure 2), MP systems were initially defined to model dynamics localized at the level of the “metabolic space”. In fact, many concepts of the MP theory, such as the concept of substance and of reaction, are inherited from that context. This, however, does not mean that MP systems cannot be a suitable framework for the modelling of other types of dynamics. In fact, MP systems have been successfully used in many different contexts: from the modelling of population dynamics to their application in the context of real function approximation [29]. More generally, MP systems can be used in each context where we want to infer models of a system from a given set of time series. In the case of gene expression analysis, MP systems are particularly convenient since we need to manage many time series coming from microarray experiments. Moreover, the discrete nature of MP models and the easy manage of substance memories introduced in [29] and widely used in [30], provide also the possibility of to naturally encode in the models the concept of regulation delay which sometimes occurs in gene networks.

Genes will be modelled here as substances whose concentration is equal to the corresponding gene expression level (usually log₂ transformed, as explained in Section 4.1). Reactions will model promotion/inhibition between genes which cause the increasing/decreasing in time of the expression levels of the involved genes. The procedure for the definition of the stoichiometry of the system is similar to the one introduced in [29] and should combine the previous knowledge about the phenomenon with suitable assumptions suggested by the shape in time of the gene expression profiles. The form of the regulators, instead, is automatically calculated by the regression algorithm LGSS which takes advantage of the defined stoichiometry and calculates new MP models with very high performance [28]. After that LGSS has provided the most adequate MP system generating the observed dynamics, we can obtain the corresponding gene network in a simple manner. In fact, there is a standard way for translating an MP grammar involving gene expressions into a corresponding gene network. In Table 1 are represented some MP reactions and their graphical representations as MP graph [25] and gene network. Since arcs in gene networks represent regulations, in general a gene network arc connects a tuner of an MP regulator with
<table>
<thead>
<tr>
<th>MP grammar</th>
<th>MP graph</th>
<th>Gene network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple promotion</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \emptyset \rightarrow \text{Gene 2}$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1}$</td>
<td>$\text{Gene 1} \rightarrow \text{Gene 2}$</td>
</tr>
<tr>
<td>Simple inhibition</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \text{Gene 2} \rightarrow \emptyset$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1}$</td>
<td>$\text{Gene 1} \rightarrow \text{Gene 2}$</td>
</tr>
<tr>
<td>Simple promotion/inhibition</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \text{Gene 2} \rightarrow \text{Gene 3}$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1}$</td>
<td>$\text{Gene 2} \rightarrow \text{Gene 3}$</td>
</tr>
<tr>
<td>Combined promotion</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \emptyset \rightarrow \text{Gene 3}$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1} + k_2 \cdot \text{Gene 2}$</td>
<td>$\text{Gene 1} \rightarrow \text{Gene 3}$</td>
</tr>
<tr>
<td>Combined inhibition</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \text{Gene 3} \rightarrow \emptyset$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1} + k_2 \cdot \text{Gene 2}$</td>
<td>$\text{Gene 1} \rightarrow \text{Gene 3}$</td>
</tr>
<tr>
<td>Comb. promotion/inhibition</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Network" /></td>
</tr>
<tr>
<td>$r : \text{Gene 3} \rightarrow \text{Gene 4}$</td>
<td>$\varphi : k_1 \cdot \text{Gene 1} + k_2 \cdot \text{Gene 2}$</td>
<td>$\text{Gene 1} \rightarrow \text{Gene 4}$</td>
</tr>
</tbody>
</table>

Table 1. Some MP reactions and their graphical representations as MP graph and gene network (arrows mean promotion, bars mean inhibition). Since arcs in gene networks represent regulations, in general a gene network arc connects a tuner of an MP regulator with the gene which is reactant (inhibition arc) or product (promotion arc) of the corresponding MP reaction. The first three reactions refer to simple promotions/inhibitions, the last three, instead, refer to combined regulations which should suggest, for example, the formation of a complex between the products of a set of genes (see the action on Gene 2 of the Complex 3-4 in Figure 2).
Table 2. The MP grammar related to the gene network of Figure 3. The corresponding gene network is reported also in the right part of the table enriched with regulator coefficients which provide a measure of the strength of the regulations (G1, G2, G3, and G4 denote Gene 1, Gene 2, Gene 3, and Gene 4 respectively).

Following the guidelines given in Table 1, we provide in Table 2 the MP grammar which is related to the gene network of Figure 3. The stoichiometry of the system and the form of the regulators give the details of the regulations which provide the observed dynamics. The form of the regulator $\varphi_2$, for example, indicates that Gene 2 is promoted by the combined action of Gene 3 and Gene 4. This fact suggests the formation, at the level of the “protein space”, of the Complex 3-4 represented in Figure 2. The stoichiometry of the rule $r_4$, instead, indicates the combined action of Gene 1 which promotes Gene 3 and, at the same time, inhibits Gene 2. Moreover, since the MP regulator coefficients can be considered here as a measure of the strength of the corresponding regulations, we can use these information in order to enrich the corresponding gene network as displayed in the right part of Table 2. This means that the MP modelling of gene networks, in addition to identifying the genes that affect a specific other gene directly, permits also to automatically estimate the strength of such effects by inferring quantitative gene networks [7].

4 From microarray raw data to MP models

Genomics and gene expression experiments are sometimes derived as “fishing expeditions” in which the goal is the individuation of new genes involved in a pathway, potential drug targets or expression markers that can be used in a predictive or diagnostic fashion [21]. The usual design for such kind of experiments is to get the time series of gene expression profiles for the entire genome of some target cells after having treated them with some specific inhibitors or some targeted up-regulators. The time series are obtained by means of microarray analysis providing the same kind of gene expression levels at different time points separated by a given time interval. The raw data will be then processed and analysed in order to get a model which explains the gene regulations which act during the experiment. In our case, the time series are used to run LGSS and obtain the MP model and the corresponding gene network of the phenomenon.
The number of the raw time series which need to be processed for a generic experiment on human cells is usually of the order of tens of thousands. Generally, however, only a small part of them are really important since many of them refer to genes whose expression profiles exhibit a pattern which is considered not related to the phenomenon under examination. For this reason, before to start with the modelling of the MP model, raw data need to be preprocessed following a methodology which comprises normalization, filtering and clustering. This methodology is given here in its main features and it has been developed during a work in progress with the Karmanos Cancer Institute, Wayne State University, Detroit. In that work, the MP theory has been successfully applied for defining the gene network underlying the regulations acting on the HER-2 oncogene-regulated transcriptome in human SUM-225 cells in order to define new therapies for the breast cancer.

4.1 Normalization and reduction of raw data

In order to reliably compare data from multiple microarray chips one needs to normalize raw data by minimizing non pertinent differences that may exist. For this reason raw data are usually $\log_2$ transformed and then normalized. The $\log_2$ transformation of the data is a standard step in gene expression analysis, since the usage of logarithms makes easier the analysis of expression ratios between genes [35]. The need for data normalization, instead, arises naturally when dealing with experiments involving multiple arrays. There are two broad characterizations that could be used for the type of variation one might expect to see when comparing arrays: interesting variation and obscuring variation. We would classify biological differences, for example large differences in the expression level of particular genes between a diseased and a normal tissue source, as interesting variation. However, observed expression levels also include variation that is introduced during the process of carrying out the experiment, which could be classified as obscuring variation. Examples of this obscuring variation arise due to differences in sample preparation (for instance labelling differences), production of the arrays and the processing of the arrays (for instance scanner differences). The purpose of normalization is to deal with this obscuring variation by reducing it as much as possible.

In literature there are many algorithms for performing normalization at the probe intensity level. Some of them are called complete data methods because they make use of data from all arrays in an experiment to form the normalizing relation. Here we propose to adopt a widely used algorithm based on quantile normalization [1] which makes the distribution of probe intensities for each array, in a set of arrays, the same. Quantile normalization is both quicker and simpler than the other complete data methods defined in literature [1]. For this reason this algorithm is particularly convenient when we need to process huge amounts of data as in the case of time series coming from microarray analysis.

In order to provide reliable measurements, the chips used for microarray analysis usually contain multiple probes developed to catch the same gene. For this reason, after having normalized the data, the number of the time series
must to be reduced in order to have only one time series for each gene analysed. Some people choose to average all the probes for a single gene for analysis, but when it is possible, it is better to follow a more sophisticated procedure which discard not reliable time series. Many software for microarray analysis associate to each gene expression measurement a detection p-value which gives a measure of its reliability. We propose to discard each time series which has the average of the detection p-values associated to its measurements greater than a threshold value. Only after this phase, we can average the remaining time series which refer to the same gene.

4.2 Data filtering

The genes selected in the previous section have normalized time series with reliable measurements. Now we need to find out among them which genes exhibit an interesting expression profile. As introduced at the beginning of the section, many times the microarray analysis are performed on target cells which have been treated with some specific inhibitors or some targeted up-regulators. In these cases, the filtering procedure has the goal of selecting only those genes whose transcription seems to be regulated by the pathway affected by the inhibitors or by the up-regulators used in the experiment. If we consider the microarray analysis as a “fishing expeditions”, the filtering procedure is the one which finally selects the “fish”, i.e. those genes involved in the pathway, which can be potential drug targets or expression markers that can be used in a predictive or diagnostic fashion [21].

The filtering procedure usually considers the variation in time of the expression level of each gene. A rough algorithm to do this consists into calculating the maximum log\(_2\) fold-change for each gene\(^1\). After that, the procedure will consider only those genes which have a log\(_2\) fold-change grater than a suitable value of threshold (usually between 1 and 2). This procedure, however, permits to remove time series which does not change at all in time (such as the one displayed in the left part of Figure 4), but many times does not remove time series such as the one displayed in the right part of Figure 4. This time series has a log\(_2\) fold-change greater than 2, but it refers to a gene which exhibits a “chaotic” expression profile. This is not desirable because genes which are regulated by the pathway under examination ought to demonstrate expression level changes that are time dependent during the experiment. In other words, they ought to exhibit a consistent pattern of change measured at several time points, and not just a large change at a single time point.

In order to examine this feature, many different filtering algorithms have been defined in literature [19] which filter gene expression profiles which have: i) small variance; ii) low absolute coefficient of variation (CV); iii) low absolute expression values; or iv) low entropy. These filters, however, are not very reliable

\(^1\) Since our time series have been log\(_2\) transformed, the log\(_2\) fold-change of one time series is given by the subtraction of the maximum expression value with the minimum one.
Fig. 4. Examples of two gene expression profiles which are not time regulated. The first one (on the left part) exhibits a dynamics which has a very low \( \log_2 \) fold-change, the second one, instead, exhibits a dynamics which has an high \( \log_2 \) fold-change (more than 2), but where expression level changes are not time dependent.

because they permit to decrease the number of false positives, but they do not explicitly test the fact that expression level changes should be time dependent. In order to examine this, we introduce here the idea of a more sophisticated procedure based on the following assumption: if some genes are regulated by the pathway under examination, then their expression profiles must change in a time dependent way and they must exhibit a dynamics which can be approximated by a polynomial model (see Figure 5) with degree 1 (linear ↗ and ↘ expression profiles), 2 (parabolic ∪ and ∩ expression profiles) or 3 (cubic ~ and ~ expression profiles). The polynomial models can be calculated by means of a standard curve fitting procedure, and then the association gene-polynomial model is carried out by means of a suitable F-test. The filtering procedure will discard all the genes which do not fit well the polynomial models. Finally, the remaining genes are filtered again by selecting those whose \( \log_2 \) fold change, calculated starting from the plot of their corresponding polynomial model, is greater than a given threshold.

Genes with linear expression profiles are those genes which are assumed to be directly regulated by the inhibitors or by the up-regulators used during the experiment. The genes with parabolic and cubic expression profiles, instead, are those which are considered to be regulated by a more complex pathway in which the action of the inhibitors/up-regulators is combined with the action of other transcript products. Recalling Figure 2, the action of the transcript products is related to some regulation loops acting on the “metabolic” and on the “protein space” (an example of such kind of regulation is given by the action of Gene 3 and of Gene 4 on the transcription of Gene 2 in Figure 2). These regulation loops act as “side effects” on the gene regulations and need careful attention since they may cause unexpected effects. For example, during the developing a new therapy for a cancer, it should be useful to study the effect on the cancerous cell transcriptome of some particular inhibitors (for example some inhibitors of
Fig. 5. Examples of the six expression profiles defined in the filtering analysis. Starting from the top-left corner we have an example of a ↗ profile, a ∪ profile, and a ∼ profile in the first row; an example of a ↘ profile, a ∩ profile and a ∼ profile in the second row.

a growth factor) in order to understand if it is possible to induce a programmed cell death of the cancerous cells. In such kind of analysis, a side effect caused by some transcript products may promote the transcription of genes which are inhibited by the drug causing a dramatical lost of effectiveness of the therapy (resistance). In these cases, the MP modelling of the gene network can help the biologist to understand what is going wrong and which countermeasures are needed.

4.3 Data clustering

The genes selected in the previous section are those we need to consider during the modelling phase. The total number of the genes is usually reduced to the 5 – 10% of the number of the genes we started from. This means that sometimes the number of the genes may be of the order of thousands, a number which is already too big if we want to singularly analyse each gene expression profile. For this reason, many times a clustering phase is needed in order to divide the genes of each polynomial class in a set of sub-classes which collect genes that have very similar expression profiles. In this way the MP modelling of the phenomenon will consider only an expression profile for each cluster (called cluster profile) calculated by averaging the time series of the gene expression profiles which are
in the same group. This approach is called in literature guilt by association [21] and assumes that genes with similar expression patterns are functionally related to each other.

In literature many different cluster algorithms have been defined, most of them especially written to solve the problem of clustering gene expression time series [6]. Since the goal of our analysis is the discovery of the mechanisms of a gene network, here we need to focus our attention on the variation of the \( \log_2 \) expression level of each gene from one time step to the following one in the time series. In fact, this amount should be directly related to the perturbing factor acting on the system during the microarray analysis.

### 4.4 From data clustering to gene networks via MP systems

The complexity of the filtering procedure described in Section 4.2 allows us to adopt a standard hierarchical clustering technique [17] which starts with only one gene in a cluster, and then gradually merges clusters into larger ones until the “distance” between clusters, computed by considering the time series of the expression profiles associated to each gene, becomes greater than a given threshold value. The kind of distance is the average value of the point-wise difference between the derivative time series of the gene polynomial models calculated as in Section 4.2. The use of derivatives is motivated by the assumption that genes having similar change rates are influenced by the same factors. Now, if we consider clusters as nodes of an MP graph according to the representation of Tables 1 and 2, then we can apply the LGSS algorithm providing the best MP grammars approximating the observed cluster profiles. Consequently we deduce the gene network identified by the provided grammars.

Finally, a sub-cluster division is defined for supporting biologists in the determination of specific genes which are more relevant for each cluster. To this end, the genes of each cluster are sorted according to the value of the Y intercept of their polynomial models because those having higher \( \log_2 \) expression level (or ranging in some value intervals) could be the genes on which much effort must to be focused in the research of new drug targets or expression markers.

### 5 Conclusions

In this paper we outlined a general methodology of application of the MP theory to gene expression analysis. After having provided some introductory concepts about the structure and the goals of a DNA microarray experiment, in Section 2 we introduced gene networks as a particular type of molecular interaction network which represents the regulation relationship which can occur between genes.

In Section 3 we introduced a representation of gene networks by means of Metabolic P systems, while in Section 4 we elaborated on a pre-processing method of raw time series coming from microarray analysis. This method is divided in three parts (normalization, filtering and clustering of data) and it
A methodology based on MP theory for gene expression analysis

has been developed during a work in progress with the Karmanos Cancer Institute, Wayne State University, Detroit, where a specific application of the MP theory has been successfully developed. In fact, the gene network underlying the regulations acting on the HER-2 oncogene-regulated transcriptome in human SUM-225 cells was analysed with the purpose of suggesting new therapies for the breast cancer. We plan to develop this approach in other cases of gene expression analysis, in order to prove its general biological validity.

References

Simulation of Spiking Neural P Systems Using Pnet Lab

Venkata Padmavati Metta¹, Kamala Krithivasan² and Deepak Garg³

¹ Bhilai Institute of Technology, Durg, India
² Indian Institute of Technology, Chennai, India
³ Thapar University, Patiala, India

Abstract. In this paper we propose a methodology for translating spiking neural P systems without delay into Petri net models. This enables us to study several aspects concerning the formal verification of spiking neural P systems, by using results and tools regarding various classes of Petri nets. A tool called Pnet Lab is used to simulate and check certain properties of these systems.

Key words: Spiking Neural P System, Petri Net, Pnet Lab.

1 Introduction

P systems (membrane systems) were introduced in [10] as parallel computing devices inspired by the hierarchical structure of membranes in living organisms, and the biological processes which take place in and between cells. Spiking neural P systems (for short, SN P systems) [4] are a class of P systems which use ideas from neural computing, more precisely the way neurons communicate with each other by sending electrical signals of identical voltage, called spikes, through synapses - links established with neighbouring neurons. The spikes of neurons look alike, but the timing and number of spikes entering a neuron determines the way the information is encoded.

SN P system is pictorially described as a directed graph where nodes represent the neurons having spiking and forgetting rules. The rules involve the spikes present in the neuron in the form of occurrences of a symbol \( a \). The arcs indicate the synapses among the neurons. The SN P system works in a locally sequential and globally maximal way. That is, each neuron, at each step, if more than one rule is enabled, then only one of them can fire. But still, all neurons fire in parallel at the system level. The configuration of the system at any time is represented by the number of spikes present in each neuron at that time. Configuration is changed by the application of rules, such a step is called a transition. Thus one can define computations (sequences of transitions from the initial configuration) which can then be interpreted as specifying a result of the operation of the SN P system. Typically, one defines a notion of successful (or halting) computation together with the output produced in such case, by considering one or more neurons as output neurons, yielding to notions of functionality and computational power of the SN P system [11], including various aspects of complexity.
For efficient formalization and to deal with the implementation correctness of SN P systems, this paper proposes a method based on Petri nets. Petri nets play an important role in the modelling, analysis and verification of concurrent systems. Petri nets [12] are graphical as well as algebraic modelling tools, applicable to a large variety of information processing systems. They consist of places and transitions. Places represent the objects which indicate the availability of resources, represented by tokens. Thus places are used to represent neurons and spikes are indicated with tokens. Transitions are actions which can occur depending upon the availability of resources and thus represent spiking and forgetting rules inside the neurons. It is worth noting that as far as the rules are concerned, SN P systems are highly concurrent in nature and Petri nets are successful modelling paradigm, which allows concurrent systems to be described in a formal yet graphical and well readable way. It is attractive to adopt Petri nets to model SN P systems. So the rich theoretical concepts and practical tools from well-developed Petri nets could be introduced in the current research of SN P systems.

There are different interactions between classes of membrane systems and Petri nets that have been investigated so far. In [5] it is provided an overview of all these investigations. In this paper we are interested in those interactions investigating the role of Petri nets as a tool to express behavioural semantics for membrane systems. This has been discussed in [5] and thoroughly investigated in [7, 6] Petri nets with localities have been introduced to represent some variants of membrane systems. In [1] some problems like producer-consumer are expressed in both formalisms in order to analyse their modelling capabilities. In [8, 9] SN P systems with delay and SN P systems with anti-spikes are translated into new variants of Petri net models. However, all these new variants of Petri nets typically lack the tools for building models, for executing and observing simulation experiments. In [2], a tool for simulating simple and extended SN P system is introduced that yields only the transition diagram showing the reachable configurations for the SN P system and it lacks of graphical step-by-step simulation of the system.

This paper introduces the direct translation of standard SN P systems without delay into Petri net models that can be simulated using existing Petri net tools. As the procedure is direct, it involves less complexity in translation and also using the notions and tools already developed for Petri nets, one can describe the internal process occurring during a computation in the SN P system in a graphical way. Perhaps the greatest advantages of Petri nets are a solid mathematical foundation and the large number of techniques being developed for their analysis. These include: reachability analysis, invariants analysis (a technique using linear algebra), transformations (including reductions) preserving desired properties, structure theory and formal language theory. We considered Pnet Lab - a Java based simulation tool for Petri nets to analyse the SN P systems. Pnet Lab allows the parallel firing of all enabled transitions after resolving the conflicts that can efficiently simulate the parallel use of rules in all neurons in each step. It also allows the user defined guard function that can encode the reg-
ular expression associated with each rule. As Pnet Lab is a Java based simulation tool, it is extremely light-weight and platform independent.

This paper is organised as follows. We start with section 2 by giving brief introduction about standard SN P system. In section 3, we discuss the Petri net model considered for translations. Using these definitions as basis in section 4, we translate an SN P system into an equivalent Petri net model that can be simulated using Pnet Lab. The section 5 gives analysis results for SN P systems through Pnet Lab.

1.1 Notation

We recall here few definitions and notations related to formal languages and automata theory.

\( \Sigma \) is a finite set of symbols called alphabet. A string \( w \) over \( \Sigma \) is a sequence of symbols drawn from \( \Sigma \). \( \lambda \) denotes the empty string. \( \Sigma^* \) is the set of all string over \( \Sigma \). \( \Sigma^* - \{ \lambda \} \) is denoted by \( \Sigma^+ \). The length of a string \( w \) is denoted by \( |w| \).

A language \( L \subseteq \Sigma^* \) is said to be regular if there is a regular expression \( E \) over \( \Sigma \) such that \( L(E) = L \). The regular expressions are defined using the following rules. (i) \( \emptyset, \lambda \) and each \( a \in \Sigma \) are regular expressions. (ii) if \( E_1, E_2 \) are regular expressions over \( \Sigma \), then \( E_1 + E_2, E_1 E_2 \) and \( E_1^* \) are regular expressions over \( \Sigma \), and (iii) nothing else is a regular expression over \( \Sigma \). With each regular expression \( E \), we associate a language \( L(E) \).

When \( \Sigma = \{ a \} \) is a singleton, then the regular expression \( a^* \) denotes the set of all strings formed using \( a \), i.e., the set \( \{ \epsilon, a, a^2, a^3, \ldots \} \). The positive closure \( a^+ = a^* - \{ \lambda \} \).

2 Spiking Neural P System

Here we are introducing the basic class of spiking neural P systems.

**Definition 2.1 (SN P system)** Mathematically, we represent a spiking neural P system of degree \( m \geq 1 \), in the form

\[ \Pi=(O, \sigma_1, \sigma_2, \ldots, \sigma_m, \text{syn}, i_0) \]

1. \( O = \{ a \} \) is the singleton alphabet (\( a \) is called spike) ;
2. \( \sigma_1, \sigma_2, \ldots, \sigma_m \) are neurons, of the form

\[ \sigma_i=(n_i, R_i), \; 1 \leq i \leq m, \]

where

(a) \( n_i \geq 0 \) is the initial number of spikes contained by the cell;
(b) \( R_i \) is a finite set of rules of the following two forms:
   i. \( E/a \rightarrow \rightarrow a; t \), where \( E \) is a regular expression over \( O \), \( r \geq 1 \), and \( t \geq 0 \).
   The number of spikes present in the neuron is described by the regular expression \( E \), \( r \) spikes are consumed and it produces a spike, which will be sent to other neurons after \( t \) time units.
ii. $a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule $E/a^s \rightarrow a; t$ of type (i) from $R_i$;
3. syn $\subseteq \{1, 2, 3, \ldots, m\} \times \{1, 2, 3, \ldots, m\}$ with $(i, i) \notin$ syn for $1 \leq i \leq m$ (synapses among cells);
4. $a_0 \in \{1, 2, 3, \ldots, m\}$ indicates the output neuron.

The rules of type $E/a^s \rightarrow a; t$ are spiking rules, and they are possible only if the neuron contains $n$ spikes such that $a^n \in L(E)$ and $n \geq r$. If $E$ is omitted then the rule is applied only if the neuron contains exactly $r$ spikes. When neuron $\sigma_i$ spikes, its spike is replicated in such a way that one spike is sent to all neurons $\sigma_j$ such that $(i, j) \in$ syn, and $\sigma_j$ is open at that moment. If $t = 0$, then the spikes are emitted immediately, if $t = 1$, then the spikes are emitted in the next step and so on. In the case $t \geq 1$, if the rule is used in step $d$, then in step $d, d + 1, d + 2, \ldots, d + t$, the neuron is closed and it cannot receive new spikes (If a neuron has a synapse to a closed neuron and sends spikes along it, then the spikes are lost; biology calls this the refractory period). In step $t + d$, the neuron spikes and becomes open again, hence can receive spikes (which can be used in step $t + d + 1$). If a neuron $\sigma_i$ fires and either it has no outgoing synapse, or all neurons $\sigma_j$ such that $(i, j) \in$ syn are closed, then the spike of neuron $\sigma_i$ is lost; the firing is allowed, it takes place, but results in no new spikes.

The rules of type $a^s \rightarrow \lambda$ are forgetting rules; $s$ spikes are simply removed (“forgotten”) when applying. Like in the case of spiking rules, the left hand side of a forgetting rule must “cover” the contents of the neuron, that is, $a^s \rightarrow \lambda$ is applied only if the neuron contains exactly $s$ spikes. In this paper we will consider systems without delays (i.e., $t = 0$ in all rules). Because all rules have the delay 0, we write them in the simpler form $E/a^s \rightarrow a$, hence omitting the indication of the delay.

**Definition 2.2** (Configuration) The configuration of the system is described by the number of spikes present in each neuron. Thus $(n_1, n_2, \ldots, n_m)$ is a configuration where neuron $\sigma_i$, $i = 1, 2, 3, \ldots, m$ contains $n_i \geq 0$ spikes. The initial configuration of the system is described by $C_0 = (n_1, n_2, n_3, \ldots, n_m)$, where $n_i$ is the number of spikes present in the $i\text{th}$ neuron.

A global clock is assumed in SN P system and in each time unit each neuron which can use a rule should do it (the system is synchronized), but the work of the system is sequential locally: only (at most) one rule is used in each neuron. The rules are used in the non-deterministic manner, in a maximally parallel way at the level of the system; in each step, all neurons which can use a rule of any type, spiking or forgetting, have to evolve, using a rule.

**Definition 2.3** (Vector rule) We define a vector rule $V$ as a mapping with domain $\Pi$ such that $V(i) = r_{ij}$, $r_{ij}$ is a spiking or forgetting rule from $R_i$. i.e. \( |V(i)| = 1 \) where $1 \leq i \leq m$. If no rule is applicable from $\sigma_i$, then $V(i) = r_{ij}$. If a vector rule $V$ is enabled at a configuration $C = (n_1, n_2, \ldots, n_m)$ then $C$ can evolve to $C' = (n'_1, n'_2, \ldots, n'_m)$, where $n'_i = n_i - lhs(V(i)) + \sum_{(i,j)\in syn} rhs(V(j))$ where $lhs(V(i))$ and $rhs(V(i))$ gives the number of spikes present in the left and right hand sides of rule $V(i)$ respectively.
Definition 2.4 (Transition) Using the vector rule, we pass from one configuration of the system to another configuration, such a step is called a transition. For two configurations \( C \) and \( C' \) of \( \Pi \) we denote by \( C \Rightarrow C' \), if there is a direct transition from \( C \) to \( C' \) in \( \Pi \).

A computation of \( \Pi \) is a finite or infinite sequence of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called reachable. Note that the transition of \( C \) is non-deterministic in the sense that there may be different vector rules applicable to \( C \), as described above.

The evolution of the system \( \Pi \) can be analysed on a transition diagram as that from Fig.1(b), because the system is finite, the number of configurations reachable from the initial configuration is finite too, hence, we can place them in the nodes of a graph and between two nodes/configurations we draw an arrow if and only if a direct transition is possible between them. A computation halts if it reaches a configuration where no rule can be used. With any computation halting or not we associate a spike train, a sequence of digits of 0 and 1, with 1 appearing in position which indicates the steps when the output neuron sends spikes out of the system. One of the neurons is considered to be the output neuron, and its spikes are sent to the environment. With any spike train we can associate various numbers which are considered as computed by the system.

Consider the graphical representation of an SN P system in Fig.1, the neurons are represented by nodes of a directed graph whose arrows represent the synapses; an arrow also exits from the output neuron, pointing to the environment; in each neuron we specify the rules and the spikes present in the initial configuration. Fig.1(a) represents the initial configuration of a non-deterministic SN P system \( \Pi \). It is formally represented as:

\[
\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \text{syn}, 3),
\]

with

\[
\sigma_1 = (2, \{a^2/a \rightarrow a, a^2 \rightarrow a, a \rightarrow \lambda\}),
\]

\[
\sigma_2 = (1, \{a \rightarrow a\}),
\]

\[
\sigma_3 = (3, \{a^3 \rightarrow a, a^2 \rightarrow \lambda, a \rightarrow a\}),
\]

and

\[
\text{syn} = \{(1,2),(2,1),(1,3),(2,3)\}.
\]

This SN P system works as follows. We have three neurons, with labels 1, 2 and 3; neuron 3 is the output neuron. Initially neuron 2 has one spike with a rule and neuron 1 has two spikes with three rules and non-determinism between its first two rules. So the initial configuration of the system is \( C_0 = <2,1,3> \). All neurons can fire in the first step, with neuron 1 choosing non-deterministically between its first two rules. Output neuron 3 sends its spike to the environment by using \( a^3 \rightarrow a \). If the neuron 1 uses the rule \( a^2/a \rightarrow a \), the system reaches the configuration \( <2,1,2> \). The two spikes in output neuron 3 are forgotten in the next step. Neurons 1 and 2 also exchange their spikes; thus, as long as neuron 1 uses the rule \( a^2/a \rightarrow a \), it retains one spike and receives a spike from neuron 2, thus reaches the same configuration again.

At any instance of time, starting from step 1, neuron 1 can choose its second rule \( a^2 \rightarrow a \), which consumes its two spikes and sends a spike to neuron 2 and 3 reaching the configuration \( <1,1,2> \). In the next step neuron 1 forgets its
spike will have one spike, reaching the configuration $< 1, 0, 1 >$. In the last step
the neuron 3 outputs a spike and the system halts.

The evolution of the system $H$ can be analysed on a transition diagram as
that from Fig.1(b), because the system is finite, the number of configurations
reachable from the initial configuration is finite too, hence, we can place them in
the nodes of a graph and between two nodes/configurations we draw an arrow
if and only if a direct transition is possible between them. The rules used in
each neuron are indicated with the following conventions; for each $r_{ij}$ only the
subscript $ij$ used and $i0$ when a neuron $i$ uses no rule.

3 Petri net

A Petri net [12] is a bipartite graph with two kinds of nodes, place nodes are
represented with circles having tokens and transition nodes are represented with
bars or boxes. The directed arcs connecting places to transitions and transitions
to places may be labelled with an integer weight, but if unlabelled are assumed
to have a weight equal to 1. Now we introduce the class of Petri nets with trans-
itions having guards, to be used in the translation.

Definition 3.1 (Petri net) A Petri net with guard is represented by $N = (P, T, A,
W, G, M_0)$, where

$P = \{P_1, P_2, P_3, \ldots, P_n\}$ is a finite, non-empty set of places.
$T = \{T_1, T_2, T_3, \ldots, T_n\}$ is a finite, non-empty set of transitions.
$A \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs which connect places with transitions and transitions with places.
W: A \rightarrow N assigns weight \( W(f) \) to elements of \( f \in A \) denoting the multiplicity of unary arcs between the connecting nodes.

\( G: T \rightarrow \{true, false\} \), the guard function maps each transition \( T_i \) to boolean expression, which specifies an additional constraint which must be fulfilled before the transition is enabled.

The initial marking \( M_0 = \{n_1, n_2, \cdots n_m\} \in P \), each \( n_i \) is the number of tokens initially associated with each place \( P_i \) and \( m \) is the number of places in the net \( N \).

A place \( P_i \) is an input (or an output) place of a transition \( T_j \) iff there exists an arc \((P_i, T_j)\) (or \((T_j, P_i)\) respectively) in the set \( A \). The sets of all input and output places of a transition \( T_j \) are denoted by \( I(T_j) = \{P_i : (P_i, T_j) \in A\} \) and \( O(T_j) = \{P_i : (T_j, P_i) \in A\} \) respectively. Similarly the sets of input and output transitions of a place \( P_i \) are denoted by \( I(P_i) = \{T_j : (P_i, T_j) \in A\} \) and \( O(P_i) = \{T_j : (P_i, T_j) \in A\} \) respectively. A place without any output transition is called output place. Output place only receives tokens but does not send any tokens to other places.

**Definition 3.2 (Marking)** A marking (state) assigns to each place \( P_i \) a non-negative integer \( k \), we say that place \( P_i \) is marked with \( k \) tokens. Pictorially we place \( k \) black dots (tokens) in place \( P_i \). A marking is denoted by \( M \), an \( m \)-vector where \( m \) is the total number of places. Sub marking of a Petri net is the marking of some of its places.

The state or marking of Petri net is changed by the occurrence of transition. When a transition is enabled, it may be fired to remove a number of tokens from each input place equal to the weight of the connecting input arc and create a number of new tokens at each output place equal to the weight of the connecting output arc.

Firing rules in the Petri net model are:

1. Transition \( T_j \) is enabled iff \( T_j \) satisfies the guard condition and its every input place has at least as many tokens as the weight of the input arcs, \( M(P_i) \geq W(P_i, T_j) \) \( \forall P_i \in I(T_j) \)

2. Upon firing the transition \( T_j \) removes number of tokens from each of its input places equal to the weight of the input arcs and deposits number of tokens into the output places equal to the weight of output arcs.

Concurrency is also a concept that Petri net systems represent in an extremely natural way. Two transitions are concurrent at a given marking if they can be fired at the same time i.e. simultaneously. The set of all transitions enabled by a marking \( M \) is denoted by \( E(M) \). When a transition fires, a token is removed from each of its input places and a token is added to each of its output places. This determines a new marking in a net, a new set of enabled transitions, and so on. An important concept in Petri nets is that of conflict. Conflict occurs between transitions that are enabled by the same marking, where the firing of one transition disables the other. A major feature of Petri nets is that they do not define in any way how and when a given conflict should be resolved, leading
to non-determinism on its behaviour.

**Definition 3.3 (Step)** A step is a set $U$ of transitions which fires at a marking $M$ after resolving conflicts and is denoted by $M[U]$. The input and output places of step $U$ are given by

$IN_N U(p) = \sum_{t \in U} W(p, t)$ and $OUT_N U(p) = \sum_{t \in U} W(t, p)$ for each $p \in P$.

A step $U$ which is enabled at a marking $M$ can be executed leading to the marking $M' = M + OUT_N U(p) - IN_N U(p)$. We denote this by $M[U] M'$. A step $U$ is a maximal step at a marking $M$ if $M[U]$ and there is no transition $t'$ such that $M[U + \{t'\}]$ and for every place $p \in P$, transition $t \in U$, $t$ can only be executed if it satisfies the guard function.

A Petri net system $N$ with maximal concurrency is such that for each markings $M$ and $M'$ if there is a step $U$ such that $M[U] M'$, then $U$ is a maximal step. In this paper we are considering only maximal concurrency semantics of the Petri nets.

A computation of a Petri net $N$ is a finite or infinite sequence of executions starting from the initial marking and every marking appearing in such a sequence is called reachable. A major strength of Petri nets is their support for analysis of many properties and problems associated with concurrent systems such as reachability, boundedness and liveness. The firing of an enabled transition will change the token distribution in a net according to the transition. A sequence of firings will result in a sequence of markings.

*Coverability tree* is a tree representation of all possible markings with initial marking as the root node and nodes as the markings reachable from $M_0$ and arcs represent the transition firing. A *reachability graph* is a graph where each node represents a Petri net marking, with arcs connecting each marking with all of its next markings. The reachability graph defines a net's state space (i.e., the set of reachable states). Reachability is a fundamental basis for studying the dynamic properties of any system. A marking $M_n$ is reachable from initial marking $M_0$ if a sequence of firings that transforms $M_0$ to $M_n$. The reachability problem for Petri net is the problem of finding if a marking $M_i$ is reachable from the initial marking $M_0$. Formally a Petri net with a given marking is said to be in deadlock if and only if no transition is enabled in the marking. A Petri net where no deadlock can occur starting from a given marking is said to be alive. A place-invariant (P-invariant) is a subset of places whose total number of tokens remains unchanged under any execution of the system. A transition-invariant (T-invariant) is a multiset of transitions whose execution in a certain order will leave the distribution of tokens unchanged. Generally Petri nets are analysed using tools to study important behavioural properties of the system like invariants, reachability, liveness, boundedness etc.

### 4 Translating an SN P System to a Petri Net

In this section, we propose a formal method to translate SN P systems without delay into Petri nets suitable for simulation using Pnet Lab.
Every neuron in the SN P system is one-to-one mapped to a place in $P$. Every rule is one-to-one mapped to a transition in $T$. Regular expressions are translated into guard functions that further control the transitions.

The bindings of transitions are found by matching incoming arc expressions with tokens marking input places and checking guard satisfaction. To describe locally sequential semantics of the SN P system, a synchronizing place is added to each place to allow at most one transition (one rule) to fire from each input place.

**SN P system without delay to Petri net**: Let $\Pi=(O, \sigma_1, \sigma_2, \ldots, \sigma_m, \text{syn}, i_0)$ be an SN P system, then the corresponding Petri net $\mathcal{N}\mathcal{L}_{\Pi} \overset{df}{=} (P, T, A, W, G, M_0)$, where

1. $P \overset{df}{=} \{P_1, P_2, \ldots, P_m, P_{m+1}, P_{11}, P_{21}, \ldots, P_{m1}\}$ is the set of places. $P_i$, $1 \leq i \leq m$ are places corresponding to the neurons $\sigma_i$ and $P_{11}$ is the synchronizing place for place $P_1$ and $P_{m+1}$ is the place corresponding to environment.

2. $T \overset{df}{=} T_1 \cup T_2 \cup \cdots T_m$ where each group of transitions $T_i$ contains a distinct transition $T_{ik}$ for every rule of $r_{ik} \in R_i$.

   for every place $P_i \in P$, $1 \leq i \leq m$ do
   
   for every transition $t = T_{ik} \in T_i$ do
   
   $G(t) \overset{df}{=} (\text{if } a^{\sigma_i} \in L(E) \text{ return true else return false})$ where $n_i = M(P_i)$
   
   $W(t, P_i) = \text{lhs}(r_{ik})$, $W(P_{11}, t) = 1$
   
   if $i = m$ then $W(t, P_{m+1}) = \text{rhs}(r_{ik})$
   
   else for every $(i, j) \in \text{syn}$, $W(t, P_j) = \text{rhs}(r_{ij})$
   
   end if
   
   $W(t, P_{11}) = 1$
   
   end for
   
   end for

3. for every place $P_i \in P$, its initial marking is $M_0(P_i) \overset{df}{=} n_i$.

To capture the very tight correspondence between the SN P system without delay $\Pi$ and Petri nets $\mathcal{N}\mathcal{L}_{\Pi}$, we introduce a straightforward bijection between configurations of $\Pi$ and markings of $\mathcal{N}\mathcal{L}_{\Pi}$, based on the correspondence between places and neurons.

Let $C = < n_1, n_2, \ldots, n_m >$ be a configuration of an SN P system $\Pi$. Then the corresponding sub marking $\phi(C)$ of $\mathcal{N}\mathcal{L}_{\Pi}$ is given by $\phi(C)(P_i) \overset{df}{=} n_i$ for every place $P_i$ where $1 \leq i \leq m$ of $\mathcal{N}\mathcal{L}_{\Pi}$.

Similarly, for any vector rule $V = (r_{1j_1}, r_{2j_2}, \ldots, r_{mj_m})$ of $\Pi$, we define a step $\xi(V)$ of transitions of $\mathcal{N}\mathcal{L}_{\Pi}$ such that $\xi(V)(T_{ij}) \overset{df}{=} r_{ij}$ for every $T_{ij} \in T$. It is clear that $\phi$ is a bijection from the configurations of $\Pi$ to the markings of $\mathcal{N}\mathcal{L}_{\Pi}$, and that $\xi$ is a bijection from vector rules of $\Pi$ to steps of $\mathcal{N}\mathcal{L}_{\Pi}$.

We now can formulate a fundamental property concerning the relationship between the dynamics of the SN P system $\Pi$ and that of the corresponding Petri net:
\( C \xrightarrow{V} C' \) if and only if \( \phi(C)[\xi(V)] > \phi(C'). \)

Since the initial configuration of \( \Pi \) corresponds through \( \phi \) to the initial sub marking of \( \mathcal{NL}_\Pi \), the above immediately implies that the computations of \( \Pi \) coincide with the concurrency semantics of the Petri net \( \mathcal{NL}_\Pi \).

The reader might by now have observed that the structure of neurons in \( \Pi \) is used in the definitions of the structure of the Petri net \( \mathcal{NL}_\Pi \) (i.e., in the definitions of places, transitions and the weight function). Let \( C \) be a configuration of \( \Pi \) and there is a vector rule \( V \) enabled at \( C \) reaching a configuration \( C' \). As there is a mapping between configuration and sub markings, \( \phi(C) \) is the sub marking of Petri net \( \mathcal{NL}_\Pi \) corresponding to the configuration \( C \) of \( \Pi \). There is a one-to-one mapping between the rules in the SN P system and transitions in Petri net. For locally sequential and globally maximal concurrency firing semantics of SN P systems, the synchronizing places \( P_{11} \) for each place \( P_i \) are created to allow at most one transition to fire from each input place. So there exists a step \( \xi(V) \) enabled at the sub marking \( \phi(C) \). After the execution of the step the system reaches the configuration \( \phi(C') \). We can prove only if part in the similar way. So
the evolution of the Petri net $NL$ is the same as the evolution of the SN $P$ system $\Pi$.

5 Simulation with Pnet Lab

Pnet Lab is a software for Petri nets that provides interactive simulation, analysis and supervision. It allows modelling and analysis of Colored Petri Nets, place-transition nets, timed/untimed. For Petri net models, the computation of Transition-invariant, Place-invariant, minimal siphons and traps, pre-incidence, post-incidence and incidence matrices and coverability tree is available. We can also write user defined guard functions. The deterministic finite automaton for the regular expression $E$ is translated into a user defined guard function that enables a transition when the number of spikes (in the form of sequence of a’s) present in the neuron is in $L(E)$. It also allows the firing of multiple transitions in a single step and resolves conflicts.

Pnet Lab manages conflicts by using the following resolution policies:

1. Predefined Scheduling order: Pnet Lab assigns a static priority to the transition in conflict, based on the order in which they have been drawn;
2. Same firing rate: transition in conflict relation have the same firing probability;
3. Stochastic firing rate: transition in conflict relation have a firing probability defined a priori by the user;

A detailed manual about Pnet Lab can be found in [13]. Fig.2 shows the Petri net model for the SN $P$ system in Fig.1 modelled using Pnet Lab. Each place is named as $pi - Pj$, where $pi$ is the place name given by the tool and $Pj$ is the place name given as per methodology discussed in the previous section. The transitions are also named in the same manner. Fig.2 also gives the invariants. In [3] it is proved that finding invariants enable us to establish the soundness and completeness of the system. The tool also outputs the coverability tree which is not shown in the figure.

Fig.3 gives the output of the step-by-step simulation of the model. The symbol $\{1\}$ in the marking column indicates the presence of a token in that place. If we consider the sub marking for the first three places (the places corresponding to the neurons), the initial marking is $<2,1,3>$ which is similar as that of the SN $P$ system in Fig.1. At this marking, after the firing of transitions $t1 - T11$, $t4 - T21$, $t5 - T31$ (corresponding to rules 11,21,31 of $\Pi$), the system reaches the next sub marking $<2,1,2>$. It stays in the same marking as long as $t1 - T11$, $t4 - T21$, $t7 - T32$ are fired. When it chooses the transition $t2 - T12$ instead of $t1 - T11$, we reach $<1,1,2>$. We can observe from the Fig.3 that the configurations reachable from initial configuration of the SN $P$ system are same as the marking reachable in the corresponding Petri net model from the initial marking. So we conclude that the Petri net model in Fig.2 accurately simulates the working of the SN $P$ system $\Pi$. 

Simulation of Spiking Neural P Systems Using Pnet Lab  391
5.1 The Behavioural Properties of SN P Systems Derived from Petri Nets

Many useful behavioural properties such as reachability, boundedness, liveness of Petri nets have been investigated. We also introduce these properties for SN P systems.

For a SN P system, we define

1. **Terminating**: the sequence of transitions between configurations of a given SN P system is finite, i.e., the computation of the SN P system always halts.
2. **Deadlock-free**: each reachable configuration enables a next step.
3. **Liveness**: it is deadlock-free and there is a sequence containing steps.
4. **Boundedness**: An SN P system is said to be k-bounded or simply bounded if the number of copies of objects in each neuron for every reachable configuration will not exceed a finite number k.

**Theorem 1.** If the Petri net for a given SN P system $\Pi$ is terminating, then the SN P system $\Pi$ is terminating.
Proof. If the SN P system is not terminating, according to the definition of termination for SN P systems, there exists an infinite step sequence. When the SN P system is encoded by the Petri net, there also exists an infinite step sequence. Every step is one-to-one mapped to a transition in the Petri net, so the sequence of transition in the Petri net is not finite. Thus, this Petri net is not terminating.

**Theorem 2.** If the Petri net for a given SN P system $\Pi$ is deadlock-free, then the SN P system $\Pi$ is deadlock-free.

**Theorem 3.** If the Petri net for a given SN P system $\Pi$ has liveness, then the SN P system $\Pi$ has liveness.

**Theorem 4.** If the Petri net for a given SN P system $\Pi$ is bounded, then the SN P system $\Pi$ is bounded.

**Proof.** The proofs of Theorem 2, 3, 4, are the same as for Theorem 1.

6 Conclusion

This paper showed how the problem of analysing SN P systems without delay can be approached by using Petri net tool. We discussed a methodology for translating SN P system into Petri net model that can be simulated using Pnet Lab - a simulation tool for Petri nets is used to study the properties like coverability tree, P-invariants and T-invariants. Our future work involves the simulation of more variants of spiking neural P systems using Petri nets.

References

Generalized Gandy-Păun-Rozenberg machines
for tile systems and cellular automata

Adam Obtułowicz
Institute of Mathematics, Polish Academy of Sciences,
Sniadeckich 8, P.O.Box 21, 00-956 Warsaw, Poland
e-mail: adamo@impan.gov.pl

Abstract. A concept of a generalized Gandy–Păun–Rozenberg machine for modelling various systems of multidimensional tile-like compartments with common parts (tile faces) of compartment boundaries by graph rewriting is introduced, where some massive parallelism of computations or evolution processes generated by these systems is respected. The representation of Gandy–Păun–Rozenberg machines by Gandy machines in [14] is extended to the case of generalized Gandy–Păun–Rozenberg machines, where the machines represented by Gandy machines are equivalent to Turing machines.

1 Introduction

The paper proposes an extension of membrane computing, cf. [16], towards modelling those systems whose underlying topology evolves in a more complicated way than by membrane division and membrane creation.

We introduce a concept of a generalized Gandy–Păun–Rozenberg machine, briefly called a generalized G–P–R machine, which is aimed to be applied for modelling various systems of multidimensional tile-like compartments (cells) with common or overlapping parts (tile faces) of compartment boundaries by graph rewriting.

The concept of a generalized G–P–R machine is a generalization of the notion of a Gandy–Păun–Rozenberg machine defined and discussed in [14], [15]. The above systems of multidimensional tile-like compartments comprise the underlying tile systems of cellular automata (see anywhere for cellular automata on multidimensional grids), of the DNA based self-assembly systems (see [18] for references), of the general self-assembly systems for certain purposes, cf. [2], [10], the tile systems appearing in tile logic, cf. [3], and in geometrical or topological programming, cf. [1], [5], [7], [8], [9], [12], [13].

The concept of a generalized G–P–R machine respects some massive parallelism of computations or evolution processes generated by the above systems of multidimensional tile-like compartments.

The main theorem of the paper shows that the representation of G–P–R machines by Gandy machines given in [14] can be extended to the case of generalized G–P–R machines, hence generalized G–P–R machines are computationally equivalent to Turing machines.
In Section 2 of the paper we recall the notion of a G–P–R machine introduced in [14] and then we define a generalized G–P–R machine and prove the main theorem of the paper.

In the Appendix the used notions of graph theory and category are presented in some details.

2 Generalized Gandy-Păun-Rozenberg machines

We recall an idea of a Gandy–Păun–Rozenberg machine, briefly G–P–R machine, introduced in [14].

The core of a G–P–R machine is a finite set of rewriting rules for certain finite directed labelled graphs, where these graphs are instantaneous descriptions for the computation process realized by the machine.

The conflictless parallel (simultaneous) application of the rewriting rules of a G–P–R machine is realized in Gandy’s machine mode (according to Local Causation Principle, cf. [6]), where (local) maximality of “causal neighbourhoods” replaces (global) maximality of, e.g. conflictless set of evolution rules applied simultaneously to a membrane structure which appears during the evolution process generated by a P system. Therefore one can construct a Gandy’s machine from a G–P–R machine in an immediate way, see [14].

For all unexplained terms and notation of category theory and graph theory we refer the reader to Appendix.

Definition 1. A G–P–R machine \( \mathcal{M} \) is determined by the following data:

— a finite set \( \Sigma_\mathcal{M} \) of labels or symbols of \( \mathcal{M} \),

— a skeletal set \( \mathcal{S}_\mathcal{M} \) of finite isomorphically perfect labelled directed graphs over \( \Sigma_\mathcal{M} \), which are called instantaneous descriptions of \( \mathcal{M} \),

— a function \( \mathcal{F}_\mathcal{M} : \mathcal{S}_\mathcal{M} \rightarrow \mathcal{S}_\mathcal{M} \) called the transition function of \( \mathcal{M} \),

— a function \( \mathcal{R}_\mathcal{M} : \text{PREM}_\mathcal{M} \rightarrow \text{CONCL}_\mathcal{M} \) from a finite skeletal set \( \text{PREM}_\mathcal{M} \) of finite isomorphically perfect labelled directed graphs over \( \Sigma_\mathcal{M} \) onto a finite skeletal set \( \text{CONCL}_\mathcal{M} \) of finite isomorphically perfect labelled directed graphs over \( \Sigma_\mathcal{M} \) such that \( \mathcal{R}_\mathcal{M} \) determines the set

\[
\widetilde{\mathcal{R}}_\mathcal{M} = \{ P \vdash C \mid P \in \text{PREM}_\mathcal{M} \text{ and } C = \mathcal{R}_\mathcal{M}(P) \}
\]

of rewriting rules of \( \mathcal{M} \) which are identified with ordered pairs \( r = (P_r, C_r) \), where the graph \( P_r \in \text{PREM}_\mathcal{M} \) is the premise of \( r \) and the graph \( C_r = \mathcal{R}_\mathcal{M}(P_r) \) is the conclusion of \( r \),

— a subset \( \mathcal{I}_\mathcal{M} \) of \( \mathcal{S}_\mathcal{M} \) which is the set of initial instantaneous descriptions of \( \mathcal{M} \).

The above data are subject of the following conditions:

1) \( V(\mathcal{G}) \subseteq V(\mathcal{F}_\mathcal{M}(\mathcal{G})) \) for every \( \mathcal{G} \in \mathcal{S}_\mathcal{M} \),

2) \( V(\mathcal{G}) \subseteq V(\mathcal{R}_\mathcal{M}(\mathcal{G})) \) for every \( \mathcal{G} \in \text{PREM}_\mathcal{M} \).
3) the rewriting rules of $M$ are applicable to $S_M$ which means that for every $G \in S_M$ the set

$$\mathcal{P}_\ell(G) = \{ h | \text{h is an embedding of labelled graphs over } \Sigma$$
with $\text{dom}(h) \in \text{PREM}_M$ and $\text{cod}(h) = G$

such that for every embedding $h'$ of labelled graphs over $\Sigma$
with $\text{dom}(h') \in \text{PREM}_M$ and $\text{cod}(h') = G$

if $\text{im}(h)$ is a labelled subgraph of $\text{im}(h')$, then $h = h'$$

of maximal applications\(^1\) $h$ of the rules $\text{dom}(h) \vdash \mathcal{R}_M(\text{dom}(h))$ of $M$ in places $\text{im}(h)$ is such that the following conditions hold:

(i) $V(\mathcal{G}) = \bigcup_{h \in \mathcal{P}_\ell(\mathcal{G})} V(\text{im}(h))$, $E(\mathcal{G}) = \bigcup_{h \in \mathcal{P}_\ell(\mathcal{G})} E(\text{im}(h))$, $h \in \mathcal{P}_\ell(\mathcal{G})$

(ii) for all $h_1, h_2 \in \mathcal{P}_\ell(\mathcal{G})$ the equation $\ell_{\mathcal{G}_{h_1}}(\hat{h}_1^{-1}(v)) = \ell_{\mathcal{G}_{h_2}}(\hat{h}_2^{-1}(v))$ holds for every $v \in V(\text{im}(h_1)) \cap V(\text{im}(h_2))$, where $\ell_{\mathcal{G}_{h_1}}, \ell_{\mathcal{G}_{h_2}}$ are the labelling functions of $\mathcal{G}_{h_1} = \mathcal{R}_M(\text{dom}(h_1))$, $\mathcal{G}_{h_2} = \mathcal{R}_M(\text{dom}(h_2))$, respectively, and $\hat{h}_1^{-1}, \hat{h}_2^{-1}$ are the inverses of isomorphisms induced by the embeddings $h_1, h_2$, respectively.

(iii) $\mathcal{F}_M(\mathcal{G})$ is a colimit of a gluing diagram $D^G$ constructed in the following way (the construction of $D^G$ is provided by (ii)):

- the set $\mathcal{I}$ of indexes of $D^G$ is such that $\mathcal{I} = \mathcal{P}_\ell(\mathcal{G}) \cup \{ \Delta \}$, where $\Delta \not\in \mathcal{P}_\ell(\mathcal{G})$ is the center of $D^G$,

- the family $\mathcal{G}_i$ ($i \in \mathcal{I}$) of labelled graphs of $D^G$ is such that $\mathcal{G}_\Delta = \mathcal{R}_M(\text{dom}(h))$ for every $h \in \mathcal{P}_\ell(\mathcal{G})$, and $\mathcal{G}_\Delta$ is such that $V(\mathcal{G}_\Delta) = V(\mathcal{G})$, $E(\mathcal{G}_\Delta) = \emptyset$, and the labelling function $\ell_{\mathcal{G}_\Delta}$ is such that provided by (ii)

$$\ell_{\mathcal{G}_\Delta}(v) = \ell_{\mathcal{G}_h}(\hat{h}^{-1}(v))$$

for every $v \in V(\text{im}(h))$ and every $h \in \mathcal{P}_\ell(\mathcal{G})$, where $\hat{h}^{-1}$ is the inverse of the isomorphism $h$ induced by the embedding $h$,

- the gluing conditions $g_h$ ($h \in \mathcal{P}_\ell(\mathcal{G})$) of $D^G$ are defined by

$$g_h = \{ (v, \hat{h}^{-1}(v)) | v \in V(\text{im}(h)) \}$$

for every $h \in \mathcal{P}_\ell(\mathcal{G})$, where $\hat{h}^{-1}$ is the inverse of the isomorphism $h$ induced by embedding $h$,

(iv) the following equations hold:

$$V(\mathcal{F}_M(\mathcal{G})) = \bigcup_{i \in \mathcal{I}} V(\text{im}(q_i))$$
and
$$E(\mathcal{F}_M(\mathcal{G})) = \bigcup_{i \in \mathcal{I}} E(\text{im}(q_i))$$

for the canonical injections $q_i : \mathcal{G}_i \to \mathcal{F}_M(\mathcal{G})$ ($i \in \mathcal{I}$) forming a colimiting cocone of the diagram $D^G$ defined in (iii),

---

\(^1\) with respect to the relation of being a labelled subgraph which can be treated as a natural priority relation between the applications of the rewriting rules
(v) the canonical injection \( q_{\Delta} : G_{\Delta} \to F_M(G) \) is an inclusion of labelled graphs, where \( \Delta \) is the center of \( D^G \) and \( q_{\Delta} \) is \( \Delta \)-th element of the colimiting cocone in (iv).

Thus \( F_M(G) \) is the result of simultaneous application of the rules \( \text{dom}(h) \vdash R_M(\text{dom}(h)) \) in the places \( \text{im}(h) \) for \( h \in P(\ell(G)) \), where one replaces simultaneously \( \text{im}(h) \) by \( \text{im}(q_{\Delta}) \) in \( G \) for \( h \in P(\ell(G)) \), respectively.

A finite sequence \( (F_M^n(G))_{i=0}^n \) is called a \textit{finite computation of} \( M \), the number \( n \) is called the \textit{time} of this computation, and \( F_M^n(G) \) is called the \textit{final instantaneous description} for this computation if

\[
F_M^0(G) = G \in I_M, \quad F_M^{n-1}(G) \neq F_M^n(G), \quad \text{and} \quad F_M(F_M^n(G)) = F_M^n(G),
\]

where \( F_M^i(G) \) is defined inductively: \( F_M^1(G) = F_M(F_M^0(G)) \).

For a computation \( (F_M^n(G))_{i=0}^n \), its \textit{space} is defined by

\[
\text{space}(M, G) = \max \{ \text{the number of elements of } V(F_M^i(G)) \mid 0 \leq i \leq n \}
\]

for \( G \in I_M \), where intuitively \( \text{space}(M, G) \) is understood as the size of hardware measured by the number of indecomposable processors\(^2\) used in the computations.

**Definition 2.** A \textit{generalized G–P–R machine} \( M \) is defined by the following data:

— the sets \( \Sigma_M, \mathcal{S}_M, I_M \) and the functions \( R_M : \text{PREM}_M \to \text{CONCL}_M, \ F_M : \mathcal{S}_M \to \mathcal{S}_M \), where \( \mathcal{S}_M, \text{PREM}_M, \text{CONCL}_M \) are skeletal sets of finite isomorphically perfect labelled directed graphs over \( \Sigma_M \), the sets \( \Sigma_M, \text{PREM}_M, \text{CONCL}_M \) are finite sets, the condition 2) holds for \( R_M \), and \( I_M \) is a subset of \( \mathcal{S}_M \);

— besides the function \( R_M \) defining \textit{rewriting rules} there is enclosed a new function \( R^a_M : \text{PREM}^a_M \to \text{CONCL}^a_M \), where \( \text{PREM}^a_M, \text{CONCL}^a_M \) are finite skeletal sets of finite isomorphically perfect labelled directed graphs over \( \Sigma_M \) and \( R^a_M \) defines \textit{auxiliary gluing rules} \( P \vdash^a C \) (\( P \in \text{PREM}^a_M, C = R^a_M(P) \)) for defining common parts of the boundaries of new compartments appearing in a step of an evolution process;

— the above data are subject of the following conditions:

A) for every \( G \in \text{PREM}^a_M \) we have \( V(G) \subseteq V(R^a_M(G)) \), the set \( P(\ell(G)) \) defined as in 3) satisfies 3)(iii), and there exists a generalized gluing diagram \( D(G) \), called \textit{gluing pattern} determined by \( G \), such that

\[ a_1 \) the set \( I(G) \) of indexes of \( D(G) \) is a set \( \{ \Delta \} \cup \hat{I}(G) \) with \( \Delta \) being the center of \( D(G) \), \( \hat{I}(G) \subseteq P(\ell(G)) \), and \( \Delta \notin \hat{I}(G) \); \]

\[ a_2 \) the family of graphs \( \hat{G}_i \) (\( i \in I(G) \)) of \( D(G) \) is such that \( V(\hat{G}_\Delta) = V(G) \), \( E(\hat{G}_h) = \emptyset \), and \( \hat{G}_h = R_M(\text{dom}(h)) \) for \( h \in \hat{I}(G) \);

---

\(^2\) The indecomposable processors coincide with urelements appearing in those Gandy machines which represent G–P–R machines in [14].
a3) the gluing conditions \( gl^3_i \) for \( i \in \mathcal{I}_{\langle \mathcal{G} \rangle} \) are such that \( gl^3_i = gl_i \) for \( gl_i \) defined as in 3)(iii) for the gluing diagram \( D^\mathcal{G} \);

a4) \( \mathcal{R}^a_M(\mathcal{G}) \) is a colimit of \( D(\mathcal{G}) \) with gluing conditions \( gl^i_j \), \( \{ i, j \} \subseteq \mathcal{I}_{\langle \mathcal{G} \rangle} \) and \( i \neq j \) such that they are unique together with \( \mathcal{I}_{\langle \mathcal{G} \rangle} \) to make \( \mathcal{R}^a_M(\mathcal{G}) \) a colimit of \( D(\mathcal{G}) \);

B) for every \( \mathcal{G} \in \mathcal{S}_M \) the following conditions hold:

\[ b_1 \] for \( \mathcal{P}^\ell(\mathcal{G}) \) defined as in 3) with \( \text{PREM}_M \) replaced by \( \text{PREM}^a_M \) and for every \( h \in \mathcal{P}^\ell(\mathcal{G}) \) and gluing pattern \( D(\text{dom}(h)) \) defined by \( \text{dom}(h) \) the set \( \text{SCP}_h = \{ h \circ h' \mid h' \in \mathcal{I}(\text{dom}(h)) \} \), called the scope of gluing pattern \( D(\text{dom}(h)) \) in place \( h \), is a subset of \( \mathcal{P}^\ell(\mathcal{G}) \) defined as in 3) for \( \mathcal{G} \) and \( \text{PREM}_M \), where \( \circ \) denotes the composition of homomorphisms of graphs;

\[ b_2 \] the set \( \mathcal{P}(\mathcal{G}) \) defined in 3) satisfies conditions 3)(i), (ii);

\[ b_3 \] the graph \( \mathcal{F}_M(\mathcal{G}) \) is a colimit of a generalized gluing diagram \( D_G \) such that

\[ (\beta_1) \] the set \( \mathcal{I} \) of indexes of \( D_G \) is the same as the set of indexes of \( D^\mathcal{G} \) given in 3)(iii), i.e. \( \mathcal{I} = \mathcal{P}(\mathcal{G}) \cup \{ \Delta \} \);

\[ (\beta_2) \] the family of graphs \( \overline{\mathcal{G}}_i \), \( i \in \mathcal{I} \) of \( D_G \) is the same as of \( D^\mathcal{G} \) defined in 3)(iii);

\[ (\beta_3) \] the gluing condition \( gl^3_i \) is \( gl_i \) defined in 3)(iii) for every \( i \in \mathcal{I} - \{ \Delta \} \);

\[ (\beta_4) \] for all \( h_1, h_2 \) with \( \{ h_1, h_2 \} \subseteq \mathcal{I} - \{ \Delta \} \) and \( h_1 \neq h_2 \) if there exists \( h \in \mathcal{P}(\mathcal{G}) \) for which \( \{ h_1, h_2 \} \subseteq \text{SCP}_h \), then the gluing condition \( gl^3_{h_1} \) of \( D_G \) is the gluing condition \( gl^3_{h_2} \) of the gluing pattern determined by \( \text{dom}(h) \) for \( h_1, h_2 \) such that \( h \circ h_1 = h_1 \) and \( h \circ h_2 = h_2 \);

\[ (\beta_5) \] if there does not exist \( h \in \mathcal{P}(\mathcal{G}) \) such that \( \{ h_1, h_2 \} \subseteq \text{SCP}_h \) for \( h_1, h_2 \) as in (\( \beta_4 \)), then the gluing condition \( gl^3_{h_1} \) of \( D_G \) is defined to be the empty set;

\[ b_4 \] the colimiting cocone \( \beta_i : \mathcal{G}_i \rightarrow \mathcal{F}_M(\mathcal{G}) \) \( i \in \mathcal{I} \) of \( D_G \) is such that

\[ (\beta_6) \] the conditions 3)(iv) and (v) hold with \( D^\mathcal{G} \) replaced by \( D_G \);

\[ (\beta_7) \] for every at least two element subset \( H \) of \( \mathcal{I} - \{ \Delta \} \) such that

\[ \bigcap_{i \in H} \left( \text{im}(\overline{\mathcal{G}}_i) \right) \neq \emptyset \]

there exists \( h \in \mathcal{P}(\mathcal{G}) \) such that \( H \) is a subset of \( \text{SCP}_h \) of gluing pattern determined by \( \text{dom}(h) \).

The gluing conditions \( gl^1_i \) of \( D_G \) defined in (\( \beta_4 \)), (\( \beta_5 \)) determine common parts of the boundaries of new compartments appearing in a step of an evolution process.

**Theorem.** Every generalized \( G-P-R \) machine \( \mathcal{M} \) is represented by a Gandy machine defined as in [14] with \( \mathcal{R}^a_M : \text{PREM}^a_M \rightarrow \text{CONCL}^a_M \) represented by the additional set of stereotypes \( T_2 \) and additional structural mapping \( G_2 \) which are defined in the same way as for \( \mathcal{R}_M \) in [14].
Proof. The condition \((\beta_7)\) provides that the condition \((3)_r\) in Principle IV in [6] holds for the Gandy machine representing \(\mathcal{M}\).

The representation in Theorem was inspired by [19].

**Example.** We present an example of a generalized G–P–R machine. The sets \(\Sigma_{\mathcal{M}^\Box}, \mathcal{S}_{\mathcal{M}^\Box}, I_{\mathcal{M}^\Box}\) and the function \(F_{\mathcal{M}^\Box}\) are defined by

\[
\begin{align*}
\sigma_{\mathcal{M}^\Box} & = \{0, 1\} \text{ and } \mathcal{S}_{\mathcal{M}^\Box} \text{ is the set of finite 2D tile configurations } \mathcal{G}_n^i \text{ over } \{0, 1\} \text{ with } n > 2 \text{ and } 0 \leq j \leq n - 2 \text{ in Lemma 6 in Appendix, where by } \\
I_{\mathcal{M}^\Box} & \text{ is the set of finite 2D tile configurations } \mathcal{G}_n^0 \text{ for } n > 2 \text{ in Lemma 5 of Appendix; } \\
F_{\mathcal{M}^\Box} : \mathcal{S}_{\mathcal{M}^\Box} \to \mathcal{S}_{\mathcal{M}^\Box} \text{ is given by }
\end{align*}
\]

The rewriting rules are the *identity rules* \(G \vdash G\) such that \(V(G) = \{0, 1\}\) with 
\(E(G) = \{(0, 1)\}\) and \(\ell_G(i) \in \{0, 1\}\) for \(i \in \{0, 1\}\) or \(V(G) = \{0\}\) with \(E(G) = \{(0, 0)\}\) and \(\ell_G(0) \in \{0, 1\}\).

Besides the identity rules there are given the following two rewriting rules
\(G^1_p \vdash G^1_p (i \in \{1, 2\})\) such that \(V(G^1_p) = V(G^1), E(G^1_p) = E(G^1)\) for \(G^1 = (0, 0) \sqcup (0, 1)^2 \cup (0, 2)\), and \(\ell_{G^1_p}((i, j)) = 1\) for \(i = 0\) and \(0 \leq j \leq 3, \ell_{G^1_p}((i, j)) = 0\) for \(i = 1\) and \(0 \leq j \leq 3; V(G^2_p) = V(G^2), E(G^2_p) = E(G^2)\) for \(G^2 = G^1 \sqcup (1, 1)\), and \(\ell_{G^2_p}((i, j)) = 1\) for all \((i, j) \in V(G^1), \ell_{G^2_p}((2, 1)) = \ell_{G^2_p}((2, 2)) = 0; V(G^2_p) = V(G^2), E(G^2_p) = E(G^2), \ell_{G^2_p}((i, j)) = 1\) for \(i = 0\) and \(1 \leq j \leq 3, \ell_{G^2_p}((0, 0)) = \ell_{G^2_p}((i, j)) = 0\) for \(i = 1\) and \(0 \leq j \leq 3; V(G^3_p) = V(G^3), E(G^3_p) = E(G^3)\) for \(G^3 = G^2 \cup (1, 0)^\Box, \ell_{G^3_p}((i, j)) = 1\) for \(0 \leq i \leq 1\) and \(1 \leq j \leq 3, \ell_{G^3_p}((i, j)) = 0\) for \(i = 2\) or \(j = 0\).

There are given two auxiliary gluing rules \(G^0_p \vdash G^0_p (i \in \{1, 2\})\) such that
\(V(G^0_p) = V(G^1), E(G^0_p) = E(G^2)\) for \(G^4 = G^1 \cup (0, 3)^\Box, \ell_{G^4_p}((i, j)) = 1\) for \(i = 0\) and \(0 \leq j \leq 4, \ell_{G^4_p}((i, j)) = 0\) for \(i = 1\) and \(0 \leq j \leq 4; V(G^5_p) = V(G^3), E(G^5_p) = E(G^5)\) for \(G^5 = G^4 \sqcup (1, 1) \sqcup (1, 2)^\Box, \ell_{G^5_p}((i, j)) = 1\) for \(0 \leq i \leq 1\) and \(0 \leq j \leq 4, \ell_{G^5_p}((i, j)) = 0\) for \(i = 2\) and \(1 \leq j \leq 3; V(G^6_p) = V(G^4), E(G^6_p) = E(G^6)\) for \(G^6 = G^5 \cup (1, 0)^\Box, \ell_{G^6_p}((i, j)) = 1\) for \(0 \leq i \leq 1\) and \(1 \leq j \leq 4, \ell_{G^6_p}((0, 0)) = \ell_{G^6_p}((i, j)) = 0\) for \(i = 1\) and \(0 \leq j \leq 4; V(G^7_p) = V(G^6), E(G^7_p) = E(G^6)\) for \(G^7 = G^6 \sqcup (1, 0)^\Box, \ell_{G^7_p}((i, j)) = 1\) for \(0 \leq i \leq 1\) and \(2 \leq j \leq 4, \ell_{G^7_p}((i, j)) = 0\) for \(i = 2\) or \(j = 0\).

For \(G^0_p\) we have
\[
\mathcal{P}(G^0_p) = \{h^1 : G^1_p \to G^0_p, h^2 : G^2_p \to G^0_p\}
\]
with \( h'(i,j) = (i,j+1) \) and \( h''((i,j)) = (i,j) \) \((i,j) \in V(G_1^{p}) = V(G_2^{p}) \), hence the gluing pattern \( D_{(G_1^{p})} \) is such that its set of indexes is \( \{\Delta\} \cup \{h', h''\} \) and its gluing condition \( gl_{h''}D \) is \( \{(2,1), (2,2)\} \).

For \( G^1_p \in S_{\mathcal{M}^\square} \) with \( 0 \leq j < n - 3 \) the embeddings \( h^1 : G^1_p \rightarrow G^i_k \) and \( h^2 : G^2_p \rightarrow G^i_k \) with \( h^1((i,k)) = (i+j+1, k+1) \) and \( h^2((i,k)) = (i+j+1, k) \) \((i,k) \in V(G^1_p) = V(G^2_p) \) belong to the scope of the gluing pattern \( D_{(G^p_{n-2})} \) in the place \( h : G^p_{n-2} \rightarrow G^i_k \) with \( h((i,k)) = (i+j+1, k) \) \((i,k) \in V(G^p_{n-2}) \). Here \( h^1 = h \circ h' \) and \( h^2 = h \circ h'' \) for \( h', h'' \) as above and hence the gluing condition \( gl_{h^1}D \) of \( G^i_k \) is the gluing condition \( gl_{h''}D' \) of \( G^i_k \) with \( \{(2,1), (2,2)\} \) of the gluing pattern \( D_{(G^p_{n-2})} \).

Since the condition \( (G_3) \) in Lemma 4 of Appendix holds for the generalized diagrams \( D_G \) \((G \in S_{\mathcal{M}^\square}) \), we get by this lemma that \( (G_7) \) also holds for these \( D_G \). Thus the above data form a generalized G–P–R machine \( \mathcal{M}^\square \).

**Open problem.** The papers [11], [18] give rise to a problem whether there exists a generalized G–P–R machine \( \mathcal{M} \) which generates Sierpiński triangles, more precisely, for an initial simple finite 2D configuration \( G \) the graph \( F^\mathcal{M}_n(G) \) is a 2D tile configuration counterpart of Sierpiński triangle for some natural number \( n \).

**Conclusion**

The paper proposes an extension of membrane computing, cf. [16], towards modelling those systems whose underlying topology evolves with natural massive parallelism of local topology transformations in a more complicated way than by membrane division and membrane creation.

The idea of a generalized G–P–R machine suggests a hypothesis that computational processes of all concurrent spatial (discrete, topological, and multidimensional) machines can be simulated with a polynomial delay by processes of transformations of labelled directed graphs according to graph rewriting rules which can be applied simultaneously to the transformed graphs, where these transformations respect common or overlapping parts of the faces of those new multidimensional tile-like compartments (cells) which appear during a transformation of a graph.

The main theorem of the paper shows how visionary was a concept of a Gandy machine by defining it by two ordered pairs \((T_1, G_1)\) and \((T_2, G_2)\) with sets \( T_1, T_2 \) of stereotypes and \( G_1, G_2 \) being structural functions (cf. [6]), where \((T_1, G_1)\) may represent a finite set of graph rewriting rules such that they respect common or overlapping parts of the faces of new tile-like compartments (cells), cf. the above hypothesis, according to a finite set of auxiliary rules represented by \((T_2, G_2)\).

Looking forward, since self-assembly is an important attribute of life, the generalized G–P–R machines, being systems equivalent to Turing machines (via
their representation by Gandy machines) and aimed to model self-assembly systems like in [18], may serve for modelling computable approximations of life, whenever it is not computable, cf. [4], [17], [20].

Appendix. Graph-theoretical and category-theoretical preliminaries

A [finite] labelled directed graph over a set $\Sigma$ of labels is defined as an ordered triple $\mathcal{G} = (V(\mathcal{G}), E(\mathcal{G}), \ell_\mathcal{G})$, where $V(\mathcal{G})$ is a [finite] set of vertices of $\mathcal{G}$, $E(\mathcal{G})$ is a subset of $V(\mathcal{G}) \times V(\mathcal{G})$ called the set of edges of $\mathcal{G}$, and $\ell_\mathcal{G}$ is a function from $V(\mathcal{G})$ into $\Sigma$ called the labelling function of $\mathcal{G}$. We drop the adjective ‘directed’ if there is no risk of confusion.

A homomorphism of a labelled directed graph $\mathcal{G}$ over $\Sigma$ into a labelled directed graph $\mathcal{G}'$ over $\Sigma$ is an ordered triple $(\mathcal{G}, h : V(\mathcal{G}) \to V(\mathcal{G}'), \mathcal{G}')$ such that $h$ is a function from $V(\mathcal{G})$ into $V(\mathcal{G}')$ which satisfies the following conditions:

$$(H_1) \ (v, v') \in E(\mathcal{G}) \text{ implies } (h(v), h(v')) \in E(\mathcal{G}') \text{ for all } v, v' \in V(\mathcal{G}),$$

$$(H_2) \ \ell_\mathcal{G}(h(v)) = \ell_{\mathcal{G}'}(v) \text{ for every } v \in V(\mathcal{G}).$$

If a triple $h = (\mathcal{G}, h : V(\mathcal{G}) \to V(\mathcal{G}'), \mathcal{G}')$ is a homomorphism of a labelled directed graph $\mathcal{G}$ over $\Sigma$ into a labelled directed graph $\mathcal{G}'$ over $\Sigma$, we denote this triple by $h : \mathcal{G} \to \mathcal{G}'$, we write $\text{dom}(h)$ and $\text{cod}(h)$ for $\mathcal{G}$ and $\mathcal{G}'$, respectively, according to category theory convention, and we write $h(v)$ for the value $h(v)$.

A homomorphism $h : \mathcal{G} \to \mathcal{G}'$ of labelled directed graphs over $\Sigma$ is an embedding of $\mathcal{G}$ into $\mathcal{G}'$, denoted by $h : \mathcal{G} \hookrightarrow \mathcal{G}'$, if the following condition holds:

$$(E) \ h(v) = h(v') \text{ implies } v = v' \text{ for all } v, v' \in V(\mathcal{G}).$$

An embedding $h : \mathcal{G} \hookrightarrow \mathcal{G}'$ of labelled directed graphs $\mathcal{G}, \mathcal{G}'$ over $\Sigma$ is an inclusion of $\mathcal{G}$ into $\mathcal{G}'$, denoted by $h : \mathcal{G} \hookrightarrow \mathcal{G}'$, if the following holds:

$$(I) \ h(v) = v \text{ for every } v \in V(\mathcal{G}).$$

We say that a labelled directed graph $\mathcal{G}$ over $\Sigma$ is a labelled subgraph of a labelled directed graph $\mathcal{G}'$ over $\Sigma$ if there exists an inclusion $h : \mathcal{G} \hookrightarrow \mathcal{G}'$ of labelled directed graphs $\mathcal{G}, \mathcal{G}'$ over $\Sigma$.

For an embedding $h : \mathcal{G} \hookrightarrow \mathcal{G}'$ of labelled directed graphs $\mathcal{G}, \mathcal{G}'$ over $\Sigma$ we define the image of $h$, denoted by $\text{im}(h)$, to be a labelled directed graph $\hat{\mathcal{G}}$ over $\Sigma$ such that $V(\hat{\mathcal{G}}) = \{h(v) \mid v \in V(\mathcal{G})\}$, $E(\hat{\mathcal{G}}) = \{(h(v), h(v')) \mid (v, v') \in E(\mathcal{G})\}$, and the labelling function $\ell_{\hat{\mathcal{G}}}$ of $\hat{\mathcal{G}}$ is the restriction of the labelling function $\ell_{\mathcal{G}'}$ of $V(\mathcal{G}')$ to the set $V(\hat{\mathcal{G}})$, i.e., $\ell_{\hat{\mathcal{G}}}(v) = \ell_{\mathcal{G}'}(v)$ for every $v \in V(\hat{\mathcal{G}})$.

A homomorphism $h : \mathcal{G} \to \mathcal{G}'$ of labelled directed graphs over $\Sigma$ is an isomorphism of $\mathcal{G}$ into $\mathcal{G}'$ if there exists a homomorphism $h^{-1} : \mathcal{G}' \to \mathcal{G}$ of labelled directed graphs over $\Sigma$, called the inverse of $h$, such that the following conditions hold:

$$(Iz_1) \ h^{-1}(h(v)) = v \text{ for every } v \in V(\mathcal{G}),$$
\((Iz_2)\) \(h(h^{-1}(v)) = v\) for every \(v \in V(G')\).

We say that a labelled directed graph \(G\) over \(\Sigma\) is isomorphic to a labelled directed graph \(G'\) over \(\Sigma\) if there exists an isomorphism \(h : G \rightarrow G'\) of labelled graphs \(G, G'\) over \(\Sigma\).

For an embedding \(h : G \rightarrow G'\) of labelled directed graphs \(G, G'\) over \(\Sigma\) we define a homomorphism \(\hat{h} : G \rightarrow \text{im}(h)\) by \(\hat{h}(v) = h(v)\) for every \(v \in V(G)\). This homomorphism \(\hat{h}\) is an isomorphism of \(G\) into \(\text{im}(h)\), called an isomorphism deduced by \(h\).

For a labelled directed graph \(G\) over \(\Sigma\), the identity homomorphism (or simply, identity of \(G\)), denoted by \(\text{id}_G\), is the homomorphism \(h : G \rightarrow G\) such that \(h(v) = v\) for every \(v \in V(G)\).

We say that a labelled directed graph \(G\) over \(\Sigma\) is isomorphically perfect if for all labelled directed graphs \(G, G'\) of \(\Sigma\) there exists a unique homomorphism \(\text{id}_G\) of \(G\) into \(G\).

**Lemma 1.** Let \(G\) be an isomorphically perfect labelled directed graph over \(\Sigma\) and let \(h : G \rightarrow G', h' : G \rightarrow G'\) be two isomorphisms of labelled graphs \(G, G'\) over \(\Sigma\). Then \(h = h'\).

We say that a set or a class \(\mathcal{A}\) of labelled directed graphs over \(\Sigma\) is skeletal if for all labelled directed graphs \(G, G'\) in \(\mathcal{A}\) if they are isomorphic, then \(G = G'\).

A gluing diagram \(\mathcal{D}\) of labelled directed graphs over \(\Sigma\) is defined by:

- its set \(\mathcal{I}\) of indexes with a distinguished index \(\Delta \in \mathcal{I}\), called the center of \(\mathcal{D}\),
- its family \(\mathcal{G}_i\) \((i \in \mathcal{I})\) of labelled directed graphs over \(\Sigma\),
- its family \(\text{gl}_i\) \((i \in \mathcal{I} - \{\Delta\})\) of gluing conditions which are sets of ordered pairs such that
  (i) \(\text{gl}_i \subseteq V(G_\Delta) \times V(G_i)\) for every \(i \in \mathcal{I} - \{\Delta\}\),
  (ii) \((v, v') \in \text{gl}_i\) implies \(\ell_{G_\Delta}(v) = \ell_{G_i}(v')\) for all \(v \in V(G_\Delta), v' \in V(G_i)\), and for every \(i \in \mathcal{I} - \{\Delta\}\),
  (iii) for every \(i \in \mathcal{I} - \{\Delta\}\) if \(\text{gl}_i\) is non-empty, then there exists a bijection
    \(b_i : L(\text{gl}_i) \rightarrow R(\text{gl}_i)\)
    for \(L(\text{gl}_i) = \{v \mid (v, v') \in \text{gl}_i\) for some \(v'\)\) and \(R(\text{gl}_i) = \{v' \mid (v, v') \in \text{gl}_i\) for some \(v\)\) such that \(\{(v, b_i(v)) \mid v \in L(\text{gl}_i)\} = \text{gl}_i\).

For a gluing diagram \(\mathcal{D}\) of labelled directed graphs over \(\Sigma\) we define a cocone of \(\mathcal{D}\) to be a family \(h_i : G_i \rightarrow G\) \((i \in \mathcal{I})\) of homomorphisms of labelled directed graphs over \(\Sigma\) (here \(\text{cod}(h_i) = G\) for every \(i \in \mathcal{I}\)) such that
\[l_G(h_\Delta(v)) = l_G(h_i(v'))\]
for every pair \((v, v') \in \text{gl}_i\) and every \(i \in \mathcal{I} - \{\Delta\}\).

A cocone \(q_i : G_i \rightarrow \tilde{G}\) \((i \in \mathcal{I})\) of \(\mathcal{D}\) is called a colimiting cocone of \(\mathcal{D}\) if for every cocone \(h_i : G_i \rightarrow \tilde{G}\) \((i \in \mathcal{I})\) of \(\mathcal{D}\) there exists a unique homomorphism
h : \tilde{G} \to G of labelled directed graphs \tilde{G}, G over \Sigma such that h(q_i(v)) = h_i(v) for every v \in V(G_i) and for every i \in \mathcal{I}. The labelled directed graph \tilde{G} is called a colimit of \mathcal{D}, the homomorphisms q_i (i \in \mathcal{I}) are called canonical injections and the unique homomorphism h is called the mediating morphism for h_i : G_i \to G (i \in \mathcal{I}).

For a gluing diagram \mathcal{D} one constructs its colimit \tilde{G} in the following way:

- V(\tilde{G}) = \bigcup_{i \in \mathcal{I}} (V_i \times \{i\}), where
  \[ V_\Delta = V(\tilde{G}_\Delta) \] for the center \Delta of \mathcal{D},
  \[ V_i = V(G_i) - R(gl_i) \] for every i \in \mathcal{I} - \{\Delta\},
- E(\tilde{G}) = \bigcup_{i \in \mathcal{I}} E_i, where
  \begin{align*}
  E_\Delta &= \{(v, \Delta), (v', \Delta) \mid (v, v') \in E(G_\Delta)\} \text{ for the center } \Delta \text{ of } \mathcal{D},
  E_i &= \{(v, i), (v', i) \mid (v, v') \in E(G_i) \text{ and } (v, v') \subseteq V_i\}
  \cup \{(v, v'), (v'', v') \in gl_i, (v', v'') \in gl_i, \text{ and } (v'', v'') \in E(G_i) \text{ for some } v'', v''\}
  \cup \{(v, v'), (v'', v') \in V_i, (v, v'') \in gl_i, (v', v'') \in E(G_i) \text{ for some } v'', v''\}
  \forall i \in \mathcal{I} - \{\Delta\},
  \end{align*}
- the labelling function \ell_{\tilde{G}} is defined by \ell_{\tilde{G}}((v, i)) = \ell_{G_i}(v) for every (v, i) \in V(\tilde{G}).

The definition of a colimiting cocone of a gluing diagram \mathcal{D} provides that any other colimit of \mathcal{D} is isomorphic to the colimit of \mathcal{D} constructed above. Hence one proves the following lemma.

**Lemma 2.** Let \mathcal{D} be a gluing diagram of labelled graphs over \Sigma. Then for every colimiting cocone q_i : G_i \to G (i \in \mathcal{I}) of \mathcal{D} if i' \neq i'', then

\[ (V(\text{im}(q_{i'})) - V(\text{im}(q_{i''}))) \cap (V(\text{im}(q_{i''})) - V(\text{im}(q_{i'}))) = \emptyset \]

for all i', i'' \in \mathcal{I} - \{\Delta\}, where \Delta is the center of \mathcal{D} and the elements of nonempty V(\text{im}(q_i)) - V(\text{im}(q_{\Delta})) with i \neq \Delta are ‘new’ elements and the elements of V(\text{im}(q_{\Delta})) are ‘old’ elements.

A generalized gluing diagram \mathcal{D} of labelled directed graphs over \Sigma is defined by:

- its set \mathcal{I} of indexes with a distinguished index \Delta \in \mathcal{I}, called the center of \mathcal{D},
- its family \mathcal{G}_i (i \in \mathcal{I}) of labelled directed graphs over \Sigma,
- its family gl_i^\Delta ((i, j)) \in \mathcal{I} \times (\mathcal{I} - \{\Delta\}) and i \neq j) of gluing conditions which are such that
  - the set \mathcal{I}^\Delta = \mathcal{I} with families \mathcal{G}_i (i \in \mathcal{I}) and gl_i^\Delta (i \in \mathcal{I} - \{\Delta\}) form a gluing diagram \mathcal{D} with \Delta as the center of \mathcal{D}.
• for every \( i \in I - \{ \Delta \} \) the set \( I^i = I - \{ \Delta \} \) with families \( G_i (i \in I - \{ \Delta \}) \) and \( g_l^j \ (j \in I - \{ i, \Delta \}) \) form a gluing diagram \( D^i \) with \( i \) as the center for \( D^i \),
• the following conditions hold:
  \((G_1) \ R(g_l^1) \cap L(g_l^1) = \emptyset \) for all \( i, j \) with \( \{i, j\} \subset I - \{ \Delta \} \) and \( i \neq j \),
  \((G_2) \ \{g_l^i\}^{-1} = g_l^0 \) for all \( i, j \) with \( \{i, j\} \subset I - \{ \Delta \} \) and \( i \neq j \), where for
  \( Q \subset A \times B \)
  \((Q)^{-1} = \{(x, y) \in B \times A \mid (y, x) \in A \times B \}\).

For a generalized gluing diagram \( D \) of labelled directed graphs over \( \Sigma \) we define a cocone of \( D \) to be a family \( h_i : G_i \to G \ (i \in I) \) of homomorphisms of labelled directed graphs over \( \Sigma \) (here \( \text{cod}(h_i) = G \) for every \( i \in I \)) such that for every \( i \in I \) the sub-family \( h_j : G_j \to G \ (j \in I^i) \) is a cocone of the diagram \( D^i \).

For a generalized gluing diagram \( D \) a colimiting cocone of \( D \), a colimit of \( D \), the canonical injections, and the mediating morphism are defined in the same way as for a gluing diagram, e.g. a cocone \( q_i : G_i \to \tilde{G} \ (i \in I) \) of \( D \) is called a colimiting cocone of \( D \) if for every cocone \( h_i : G_i \to \tilde{G} \ (i \in I) \) of \( D \) there exists a unique homomorphism \( h : \tilde{G} \to G \) of labelled directed graphs \( \tilde{G}, G \) over \( \Sigma \) such that \( h(q_i(v)) = h_i(v) \) for every \( v \in V(G_i) \) and for every \( i \in I \).

**Lemma 3.** Let \( D \) be a generalized gluing diagram with finite set \( I \) of its indexes and with center \( \Delta \), such that the following condition holds:

\((G_3) \) for all \( i, i', j \in I - \{ \Delta \} \) if \( i \neq i' \), then \( L(g_l^i) \cap L(g_l^{i'}) = \emptyset \).

Then one constructs a colimit of \( D \) to be a labelled directed graph \( \tilde{G} \) which is determined by an arbitrary nonrepetitive sequence \( i_1, \ldots, i_{n_0} \) of elements of \( I - \{ \Delta \} = \{i_1, \ldots, i_{n_0} \} \) and which is defined in the following way:

\[ V(\tilde{G}) = \bigcup_{i \in I} (V_i \times \{i\}) \text{, where } V_\Delta = V(G_\Delta), V_i = V(G_i) - R(g_l^i) \text{, for every } k \text{ with } 1 < k \leq n_0 \]

\[ V_{ik} = V(G_{ik}) - \left( R(g_l^{i_k}) \cup \bigcup_{1 \leq m < k} L(g_l^{i_m}) \right). \]

\[ E(\tilde{G}) = \bigcup_{i \in I} E_i \text{, where } E_\Delta = \{(v, \Delta), (v', \Delta) \mid (v, v') \in E(G_\Delta)\}, \]

for every \( i \in I - \{ \Delta \} \)

\[ E_i = E_1^i \cup E_2^i \cup E_3^i \cup E_4^i \text{ for } \]

\[ E_1^i = \{(v, i), (v', i) \mid (v, i), (v', i) \subset V(\tilde{G}) \text{ and } (v, v') \in E(G_i)\}, \]

\[ E_2^i = \{(v, k), (v', j) \mid (v, k), (v', j) \subset V(\tilde{G}) \text{, } i \notin \{k, j\} \subset I, \}

\[ (v, v') \in g_l^k (v', v'') \in g_l^j \text{, and } (v'', v') \in E(G_i) \text{ for some } v'', v''' \}, \]

\[ E_3^i = \{(v, i), (v', j) \mid (v, i), (v', j) \subset V(\tilde{G}) \text{, } i \neq j \in I, \}

\[ (v', v'') \in g_l^j \text{, and } (v, v'') \in E(G_i) \text{ for some } v'', v''' \}, \]

\[ E_4^i = \{(v, j), (v', i) \mid (v, j), (v', i) \subset V(\tilde{G}) \text{, } i \neq j \in I, \}

\[ (v', v'') \in g_l^i \text{, and } (v, v'') \in E(G_i) \text{ for some } v'', v''' \}. \]
the labelling function $\ell_{\tilde{G}}$ is defined by $\ell_{\tilde{G}}((v,i)) = \ell_{G_i}(v)$ for every $(v,i) \in V(\tilde{G})$.

Proof. Since by (G3) for all $i \in I - \{\Delta\}$ and $v \in V(G_i) - V_i$ there exists a unique ordered pair $(v^*,i^*) \in V(\tilde{G})$ such that $(v^*,v) \in \text{gl}_i$, one defines the $i$-th component $q_i : G_i \to \tilde{G}$ ($i \in I - \{\Delta\}$) of colimiting cocone by

$q_i(v) = \begin{cases} (v,i) & \text{if } v \in V_i, \\ (v^*,i^*) & \text{otherwise}. \end{cases}$

Lemma 4. Let $D$ be a generalized gluing diagram with finite set $I$ of its indexes and with center $\Delta$, such that the condition (G3) holds and let $q_i : G_i \to \tilde{G}$ ($i \in I$) be a colimiting cocone of $D$. Then for every $H \subseteq I - \{\Delta\}$ if

$$\bigcap_{i \in H} \left( V(\text{im}(q_i)) - V(\text{im}(q_{\Delta})) \right) \neq \emptyset,$$

then $H$ has at most two elements and if $H = \{i,i'\}$ with $i \neq i'$, then $\text{gl}_{i'}$ is nonempty.

Proof. The lemma is a consequence of Lemma 3 and the fact that two different colimits of a generalized gluing diagram are always isomorphic labelled graphs.

For two directed graphs $G_1 = (V(G_1),E(G_1))$, $G_2 = (V(G_2),E(G_2))$, we define their union by

$$G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2)).$$

For an ordered pair $(i,j)$ of integers we define $(i,j)$-tile, denoted by $(i,j)^\square$, to be a directed graph whose set of edges is such that

$$E((i,j)^\square) = \{(i,j),(i+1,j)),((i,j),(i,j+1)),((i,j+1),(i+1,j+1)),((i+1,j),(i+1,j+1)),((i,j),(i+1,j))\},$$

$V((i,j)^\square)$ does not contain any other element than that ordered pair of integers which determines some edge in $E((i,j)^\square)$.

By a finite 2D tile configuration over $\Sigma$ we understand a finite labelled directed graph $G$ over $\Sigma$ such that

(Tc1) $V(G)$ is a finite set of ordered pairs of integers including $(0,0)$,

(Tc2) $E(G) = E(\bigcup\{(i,j)^\square \mid V((i,j)^\square) \subseteq V(G)\}) \cup \{(0,0),(0,0)\}$,

(Tc3) $G$ is a connected graph.

Lemma 5. The set of all finite 2D tile configurations over $\Sigma$ is a skeletal set of isomorphically perfect labelled directed graphs over $\Sigma$. 

---

Adam Obtulowicz
Proof. We prove that a finite 2D tile configuration $G$ is an isomorphically perfect labelled directed graph by induction on the number of elements of $\{(i,j)\square | V((i,j)\square) \subseteq V(G)\}$.

We use the following directed graphs and labelled directed graphs:

— for a natural number $n \geq 1$ we define
\[
K^0_n = \bigcup \{(0,i)\square \cup (1,i)\square | \text{i is a natural number such that } 0 \leq i \leq n - 1\},
\]

— for natural numbers $j, n$ with $1 \leq j \leq n - 2$
\[
K^j_n = K^{j-1}_n \cup \bigcup \{(j+1,i)\square | \text{i is a natural number such that } 0 \leq i \leq n - j - 1\},
\]

then for natural numbers with $0 \leq j \leq n - 2$ we define a labelled directed graph $G^j_n$ over $\Sigma = \{0, 1\}$ such that
\[
V(G^j_n) = V(K^j_n), \quad E(G^j_n) = E(K^j_n) \cup \{(0,0), (0,0)\},
\]
and the labelling function $\ell_{G^j_n}$ is defined by
\[
\ell_{G^j_n}((k,l)) = \begin{cases} 0 & \text{if } k = j + 2 \text{ or } l = 0, \\ 1 & \text{otherwise.} \end{cases}
\]

Lemma 6. The set of labelled directed graphs $G^j_n$ over $\{0, 1\}$ for natural numbers $j, n$ with $0 \leq j \leq n - 2$ is a set of finite 2D tile configurations over $\{0, 1\}$.

References


Sequentiality Induced by Spike Number in SNP Systems: small universal machines

Andrei Păun

Department of Computer Science, Louisiana Tech University, Ruston
PO Box 10348, Louisiana, LA-71272 USA apaun@latech.edu

Departamento de Inteligencia Artificial,
Facultad de Informática, Universidad Politécnica de Madrid - UPM,
Campus de Montegancedo s/n, Boadilla del Monte
28660 Madrid, Spain

Bioinformatics Department, National Institute of Research and Development
for Biological Sciences, Splaiul Independenței, Nr. 296, Sector 6, Bucharest,
Romania apaun@fmi.unibuc.ro

Abstract. In this paper we consider sequential SNP systems where the sequentiality of the system is induced by the max-spike: the neuron with the maximum number of spikes out of the neurons that can spike at one step will fire. This corresponds to a global view of the whole network that makes the system sequential. We continue the study in the direction of max-spike and show that systems with 90 neurons are universal. This improves a recent result in the area.

1 Introduction

We continue the work in the direction of SNP systems based on max-spike. This type of systems was introduced in [4] and considered in [3]. In the following we give a brief motivation for the consideration of this type of devices, for more details we refer the interested reader to the previous papers in the area.

Here we consider the spiking restriction on neurons in the following way: if at any step there are more than one neuron that can spike (according to their pre-defined rules) then only the neuron(s) containing the maximum number of spikes (among the currently “active” neurons) will fire. This is contrasting with the maximal parallel application of the rules case, in which case all the “active” neurons will fire at that step. If there is a tie for the maximum number of spikes stored in the active neurons, then all the neurons containing the maximum will fire.
The main motivation behind this spiking strategy is the observation that in a population of cells of the same type (neurons in this case) which have similar types of rules (the spiking rules in our case) one can notice that the cells containing larger numbers of a specific molecule species are more active (spike faster/more frequently) than the cells containing less numbers of the same molecule. Another observation is the fact that the neurons that receive a large number of spikes are more probable to spike than the neurons that do not receive many spikes. The same modeling path was taken also when the integrate-and-fire models were defined for neurons, which leads to the neurons that receive more spikes to fire faster than the neurons that receive lower numbers of spikes.

The restriction proposed above makes the spiking of the neurons in the system almost sequential (more than one neuron can spike only in the special case when there is a tie in the number of spikes contained, the two or more active neurons that contain the maximum number of spikes over all the active neurons at that step will spike). Because of this, we will call this application strategy pseudo-sequential with respect to maximum. One can also consider the sequential strategy which resolves the ties by choosing for the next spiking neuron nondeterministically one of the neurons containing the maximum number of spikes at that moment (out of the active neurons). This second strategy will be called from now on sequential with respect to maximum.

It was already considered in [3] the difference between these devices from the point of view of generators versus acceptors. Specifically, we observed a major difference between systems with deterministic neurons working as generators as opposed to acceptors. We see that the acceptors are universal whereas the generators are only able to generate one single value (thus are non-universal).

2 Basic description and definitions

The spiking neural P systems (for short, SNP) were recently introduced in [6], and then intensely investigated, we mention only a few papers that looked at the SNP systems and are relevant for the current work: [14] and [15], thus incorporating in membrane computing [13] ideas from spiking neurons, see, e.g., [1], [8], [9]. One should note that the sequential SNP systems could be used to model also the distributed architectures (or even failures rates) from other areas of Computer Science, for example in reliability of high performance computing: [2], [11].

We now give a more detailed description of the SNP; such a system consists of a set of neurons placed in the nodes of a graph, representing synapses. The neurons send signals (spikes) along synapses (edges of the graph). This is done by means of firing rules, which are of the form $E/a^c \rightarrow a; t$. The system is represented as a directed graph consisting of a set of neurons (nodes of a graph) connected by synapses (directed edges of the graph). The neurons send signals (spikes) along these synapses by means of firing rules, which are of the form $E/a^c \rightarrow a; t$, where $E$ is a regular expression, $c$ is the number of spikes consumed by the rule that spikes a single $a$, and $t$ is the delay between firing the rule and
emitting the spike. A rule can only be used if the number of spikes in the neuron are “covered” by expression $E$, in the sense that the current number of spikes in the neuron, $n$, is such that $a^n$ is contained in the set $L(E)$ denoted by the expression $E$.

In the time interval between firing a rule and emitting the spike, the neuron is closed/blocked – it does not receive other spikes and cannot fire. After the time interval, the neuron is again open and can again fire and receive other spikes. There are also rules for forgetting spikes, of the form $a^s \rightarrow \lambda$ (s spikes are just removed from the neuron). *In this paper, for convenience, we will also refer to the forgetting rules as firing rules.* Starting from a fixed initial distribution of spikes in the neurons (initial configuration) and using the rules in a synchronized manner (a global clock is assumed), the system evolves. A computation is a sequence of transitions starting from the initial configuration. A transition is maximally parallel in the sense that all neurons that are fireable must fire. However, in any neuron, at most one rule is allowed to fire. Details can be found in [6].

An SNP can be used as a computing device in various ways. Here, as in previous papers, we will use them as generators of numbers.

In this paper, we study SNPs operating in sequential and pseudo-sequential mode as described above. Informally, this means that at each step of the computation, if there is at least one neuron with at least one rule that is fireable, we only allow to fire the neuron(s) that is(are) fireable and contain the maximum number of spikes; and for each neuron firing only one spiking rule (nondeterministically chosen) is to be fired.

For the basic definitions and prerequisites we refer the interested reader to [16], [13], and [19]. We will use in the following universality proofs the fact that register machines are universal, but due to the space limitations we will not provide the prerequisite description of the register machines, the reader is referred to [19] as this is a common proof technique.

3 Spiking Neural P Systems

The original definition of spiking P systems was given in [6]; the interested reader can find in the reference above the motivation, basic results etc.. Let us recall the basic definition in what follows.

A *spiking neural membrane system* (abbreviated as SNP), of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_0),$$

where:

1. $O = \{a\}$ is the singleton alphabet ($a$ is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ are *neurons*, of the form
   $$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$
   where:
   a) $n_i \geq 0$ is the initial *number of spikes* contained in $\sigma_i$;
   b) $R_i$ is a finite set of *rules* of the following two forms:
(1) $E/a^c \rightarrow a; d$, where $E$ is a regular expression over $a$, $c \geq 1$, and $d \geq 0$;
(2) $a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from $R_i$, we have $a^s \notin L(E)$;
3. $\text{syn} \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin \text{syn}$ for $1 \leq i \leq m$ (synapses between neurons);
4. $i_0 \in \{1, 2, \ldots, m\}$ indicates the output neuron (i.e., $\sigma_{i_0}$ is the output neuron).

The rules of type (1) are firing (we also say spiking) rules, and they are applied as follows. If the neuron $\sigma_i$ contains $k$ spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \rightarrow a; d$ can be applied. The application of this rule means consuming (removing) $c$ spikes (thus only $k - c$ remain in $\sigma_i$), the neuron is fired, and it produces a spike after $d$ time units (as usual in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized). If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted at the next step, etc. If the rule is used at the step $t$ of the computation and $d \geq 1$, then we have the following setting: at steps $t, t + 1, t + 2, \ldots, t + d - 1$ the neuron is closed (this corresponds to the refractory period from neurobiology), so that it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, that particular spike is lost). At the step $t + d$, the neuron spikes and becomes again open, so that it can receive spikes (which can be used starting with the step $t + d + 1$).

The rules of type (2) are the forgetting rules; they are applied as follows: if the neuron $\sigma_i$ contains exactly $s$ spikes, then the rule $a^s \rightarrow \lambda$ from $R_i$ can be used, meaning that all $s$ spikes are removed from $\sigma_i$.

If a rule $E/a^c \rightarrow a; d$ of type (1) has $E = a^c$, then we will write it in the following simplified form: $a^c \rightarrow a; d$.

In each time unit, if a neuron $\sigma_i$ can use one of its rules, then a rule from $R_i$ must be used. Since two firing rules, $E_1/a^{c_1} \rightarrow a; d_1$ and $E_2/a^{c_2} \rightarrow a; d_2$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron, and in that case, only one of them is chosen non-deterministically. Note however that, by definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

Thus, the rules are used in the sequential manner in each neuron, but neurons were previously considered to function in parallel with each other. It is important to notice that the applicability of a rule is established based on the total number of spikes contained in the neuron. Thus, e.g., if a neuron $\sigma_i$ contains 5 spikes, and $R_i$ contains the rules $(aa)^* / a \rightarrow a; 0$, $a^3 \rightarrow a; 0$, $a^2 \rightarrow \lambda$, then none of these rules can be used: $a^5$ is not in $L((aa)^*)$ and not equal to $a^3$ or $a^2$. However, if the rule $a^3 / a^2 \rightarrow a; 0$ is in $R_i$, then it can be used: two spikes are consumed (thus three remain in $\sigma_i$), and one spike is produced and sent immediately ($d = 0$) to all neurons linked by a synapse to $\sigma_i$, and the process continues.

One can associate a set of numbers with $H$ in several ways. We follow here the idea of [6] and we consider the intervals between the very first two con-
secutive spikes of the output neuron as numbers computed by a computation. Furthermore, we will consider only halting computations.

In the following we will consider another methodology for rule application: the neuron that contains the most spikes at one moment is the next neuron firing. This will make the system sequential, thus we will call this "sequentiality based on maximum".

**Definition 1.** 1. SNP systems defined as above are working in the max sequentiality manner if (by definition) the system is choosing as the spiking neuron at each step only one of the neurons that can fire, and furthermore, the spiking neuron chosen at each time-step has the maximum number of spikes stored among all the other active neurons in that step.

2. Systems can work in max pseudo-sequentiality manner if (by definition) at each time-step fire all the neurons that store the maximum number of spikes among all the active neurons at that step.

Of course max sequentiality is forcing the system to work in a sequential manner since at most one neuron can fire at each step, whereas the max pseudo-sequentiality allows two or more neurons to fire at the same time if all those neurons hold the exactly same number of spikes and that number is the highest value of spikes that is stored among all the active neurons at that moment.

We pass now to give the results of the paper.

### 4 Max pseudo-sequentiality result

We will start the description of results of these systems by giving the first theorem about systems based on max pseudo-sequentiality with delays. We will show the universality of such extended systems using 104 neurons as opposed to the result in [5] where the strongly sequential were shown to be not universal. In [3] the construction needed for such systems used one neuron for each register, one neuron for each instruction label, 4 extra neurons for the ADD instructions and 3 extra neurons for the SUBtract instructions. If one is using exactly that construct and follows the result from [12] by simulating an universal machine with a small number of instructions then (with the modifications suggested in [12] and some preprocessing) the result would become: 9 registers, 25 instructions, 10 ADD instructions and 14 SUB instructions thus overall it would require at least 116 neurons not considering the initialization and halting phase that together would require another roughly 9 neurons making the number 125. We improve this result in the following by considering an improvement in the ADD module and reaching 104 neurons in the universal system; if we consider also the extended rules (when a neuron spikes it can send more than one spike to the other neurons) then the number of neurons in this case is reduced to 90.

**Theorem 1.** Unbounded SNP systems in the pseudo-sequential mode operating in max sequentiality mode with delays are universal with at least 104 neurons.
Proof. We will show that Korec’s register machine [7] can be simulated by a system working in a Max sequentiality manner, with rules using delays. As we will see later on, we will need the delays only for the ADD rules in the register machine. That machine has 8 registers and 23 instructions including the halt instruction, we will add one more register that is non decreasing (and this will be the output register) and to integrate it we will replace the old halt instruction \( l_h : \text{HALT} \) with the instructions \( l_h : (\text{SUB}(0), l_{22}, l'_{h}) \), \( l_{22} : (\text{ADD}(8), l_h) \), and \( l'_{h} : \text{HALT} \). Thus at the end of the program of the initial machine we will copy the contents of the old output register (register 0) into the register 8 which will serve as the new output register. We do this so that no subtraction rule will be applicable to the output register and we will be able to use the same proof technique as used before in [3].

Let us give a brief description of the construction: In the system we will have neurons associated with each label in the program code of the register machine, also the registers will be modeled by neurons holding \( 2^n \) spikes for the value \( n \) being stored in the register. Thus the ADD module will increase by 2 the number of spikes stored in the neuron associated with the register \( r \) (effectively incrementing the register \( r \) in the simulation) and then choose nondeterministically a new instruction to execute out of the two possibilities given by the ADD instruction.

In the end we will give a module INIT that will take care of initializations, for now let us assume that the system starts with \( 2^n \) spikes in neuron 1 for the recursive function \( g(x) \) and \( 2m \) spikes in neuron 2 codifying \( y \), where \( g(x) \) and \( y \) are defining the partial recursive function that will be computed from the list of a fixed admissible enumeration of the partial recursive functions: \( \phi_0, \phi_1, \ldots \). I.e. \( \phi_x(y) \) is computed by the register machine \( M \) that we simulate. Actually we computer the double of the \( \phi_x(y) \) in register 8 and then in the HALT module we show how to output the result.

In the following we give the neurons necessary to simulate an \( l_1 : (\text{ADD}(r)l_2) \) rule (note that we are in the acceptor case, thus the nondeterminism at the level of the ADD rules is not needed):

The module works as follows: the neuron \( l_1 \) spikes, signalling that the instruction \( l_1 \) is being executed, then the neurons \( a_1 \) and \( a_2 \) are activated, since \( a_1 \) has at this moment two spikes and \( a_2 \) only one, \( a_1 \) fires first, but has a delay of size one associated with its rule, at the next step \( a_2 \) fires (since at that moment it is the only neuron fireable), making the spikes from \( a_1 \) and \( a_2 \) to arrive at the same time in the neuron \( r \). We will see later that the neurons of type \( r \) can only fire when they hold an odd value, thus receiving two spikes keeps the neuron \( r \) inactive. At the same time \( a_2 \) sends a spike towards \( l_2 \). This we continue to simulate the work of the register machine going into the instruction \( l_2 \) as was needed.

We will now give the module simulating the \( \text{SUB} \) instruction from the register machine. We show how we are simulating all the ADD and SUB instructions in general.
When the neuron $l_1$ fires, it sends two spikes, one in the neuron $r$ (modeling the register that is decremented or checked for zero) and one in the neuron $s_1$. We note that we will start with two spikes in the neurons modeling the registers (we will take care of the correct counting in the finalizing module), we also start with three spikes in the neuron $s_1$ thus at the next step the neuron $s_1$ contains exactly 4 spikes, whereas the neuron $r$ contains exactly $2n + 3$ spikes, where $n$ is the contents of the register $r$ in the counter automaton.

We have now two possible cases:

Case I: if the register $r$ is empty, it means that the neuron $r$ holds exactly 3 spikes. Since these are the only two neurons that can fire at this moment ($r$ and $s_1$), then $s_1$ will execute first since it has four spikes (one more than $r$). This means that all four spikes in neuron $s_1$ are deleted through the forgetting rule, then at the next step $r$ spikes sending one spike back to $s_1$ and activating $s_2$. At the next step $s_2$ fires sending one more spike in $s_1$ and sending another one back to $r$ which was empty. At the next step $s_3$ fires also replenishing the two initial spikes in $r$ and the third spike in $s_1$. This means that we reached a configuration similar to the original configuration when $l_1$ spiked, and now $l_3$ is activated (since the register was empty).

Let us consider the case when the register $r$ would be non-empty: Case II: then $r$ would hold $2n + 3$ spikes, with $n \geq 1$, thus $r$ will hold at least 5 spikes, more than the four held by $s_1$. Thus $r$ spikes sending one spike to $s_1$ and another
spike to $s_2$. At the next step $s_1$ will have 5 spikes as opposed to $s_2$ that holds only one, thus $s_1$ spikes removing two spikes. That means that at the next step we will have $s_1$ holding 3 spikes and being inactive, $s_2$ holding 2 and $l_2$ holding one. Thus next $s_2$ will forget its two spikes, making the configuration as before and then the simulation can continue with $l_2$.

It is clear that the rules from the register machine are correctly simulated by the modules presented above. What remains is the initial module that adds $2 \ast g(x)$ in neuron 1 and $2 \ast y$ in neuron 2, starts the $l_0$ instruction and also the finishing stage in which the output register (register 8) is read and processed in our setting:

The INIT module functions in the following way: the input neuron (neuron $i_0$) receives exactly 3 spikes, the distance in clock cycles between the first two is the value $g(x)$ that needs to be put in the register 1 and the distance between the spike 2 and 3 is the value $y$ that will be put in register 2. Neurons $i_1$ and $i_2$ are both active immediately after the first spike received from $i_0$ and will put at each time-step two spikes into register 1. This is because we work in pseudo-sequentiality mode and both $i_1$ and $i_2$ are having the maximum spikes thus both spike simultaneously. When the second spike comes into $i_0$ this neuron will spike because it has the most spikes in the system, it will increase the number of spikes in $i_1$ and $i_2$ stopping them, but it will activate the neurons $i_3$ and $i_4$ that will push into register 2 the same way as the neurons $i_1$ and $i_2$ into register 1. When the third spike comes in neuron $i_0$ then also $i_3$ and $i_4$ will be de-activated and now $i_5$ will be activated making the instruction $l_0$ active by sending a spike there. This finishes the INIT module which will obviously perform the required actions: put a double amount into registers 1 and 2 and activate instruction $l_0$.

Let us now pass to the HALT module. From the construction we know that the output register (register 8) is never decrementing.
When we activate the halting label in the register machine we send a spike in the output neuron (the neuron $s_1$ in the picture above). Thus at the next step $s_1$ spikes (being the only active neuron), then both $r$ and $s_1$ are active. Thus the one holding the maximum number of spikes will fire. One can notice that in $r$ we are deleting exactly 2 spikes each time, thus $s_1$ will let $r$ spike as long as the register $r$ (in the register machine) is non-empty, and at each time step two more spikes are removed (thus the register $r$ is decremented by one each clock cycle). Thus the second time that $s_1$ spikes would have been exactly $n$ clock cycles after the first spike, making the whole system to correctly simulate the work of the starting register machine.

It remains to count now the neurons: INIT module requires 6 neurons plus the registers (green) and the start instruction label (grey). HALT requires 2 neurons, we will have 9 neurons for the registers (green) and 25 neurons (grey) for the instruction labels (the 23 instructions from the modified Korec construction plus the one ADD and one SUB instructions that deal with our new register 8). So we will have exactly 10 ADD, 14 SUB and 1 halt instructions. Since in the ADD
module we need 2 extra neurons \((a_1, a_2)\) and in the SUB module we need 3 
\((s_1, s_2, s_3)\) the total becomes: \(6 + 9 + 25 + 10 \times 2 + 14 \times 3 + 2 = 104\).

This completes the proof. \(\square\)

If we would consider the output of the system to be a special neuron, then the halting module will have one neuron less (neuron \(s_1\)) and the output of the machine would be found in the rightmost neuron in the halting module. This in that case one neuron would be saved.

If we consider the case of extended systems (where the neurons can send more than one spike through the synapse in one clock cycle), then we can easily remove the delay that appears in the \(\textit{ADD}\) module and we can improve the result by decreasing the number of extra neurons needed in the SUB case to 2 from 3.

\textbf{Theorem 2.} There are extended \(\textit{SN P}\) systems with max sequentiality with 90 neurons which are universal.

\textit{Proof.} We will change both the INIT and the SUB modules. INIT will be changed in the following sense: neuron \(i_1\) will have a spike in the initial configuration, and their rules will be \(i_1 : a^2 \rightarrow a^2, 0\) and \(i_2 : a^3|a^2 \rightarrow a^2, 0\). In this way they will fire alternatively until they are stopped by the next spike from \(i_0\). The same will happen also for the neurons \(i_3\) and \(i_4\). Thus we will work in the strongly sequential mode and be able to do the initializations.

We change the \(\textit{SUB}\) module in the following way: change the rule in \(\textit{r}\) to: \(a^{2k+3}|a^3 \rightarrow a^3\), in \(s_1\) the rule \(a^3|a^2 \rightarrow a\) will be replaced with \(a^7|a^4 \rightarrow a\), then the synapses \((s_1, s_2)\) and \((s_2, s_3)\) will be deleted, \(s_3\) will be deleted and we will have synapse from \((s_2, l_3)\). All these changes are depicted in the following figure.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure5.png}
\caption{The subtract in extended mode module for \(l_1 : (\text{SUB } r, l_2, l_3)\)}
\end{figure}
Basically we re-use the previous construction eliminating the neuron \( s_3 \) that was used to replenish \( s_1 \) and \( r \). To achieve this we have now \( s_2 \) sending two spikes to \( r \) and \( r \) sending three spikes to \( s_2 \).

This construction reduces the number of auxiliary neurons necessary in the SUB instruction from 3 to 2 making the total number reach: 104-14=90 neurons. Everything else remains the same.

\[ \square \]

5 Final Remarks

We plan to continue the investigation of this special type of sequentiality; it would be interesting if in the case of strong sequentiality we could obtain a similar result. By intuition, we do not think so as at each time-step only one neuron would spike, thus at most \( n \) spikes can reach a register during the \( n \) clock-cycles that it takes the input neuron to receive the first value, and when the second value comes it would be a small complication to deal with that result. Probably the input module will “grow” to deal with these issues in that case. We are looking also at the min-spike case. Finally we need to mention the work of Pan and collaborators in [17], [18] which devised another proof technique that could prove useful in improving the current results.

Acknowledgements

We gratefully acknowledge support in part from CNCS grant TE-92, BIODIV 105/2011, NSF CCF-1116707.

References

17. X. Zhang, X. Zeng, and L. Pan: Smaller universal spiking neural P systems, Fundamenta Informaticae, 87 (1), (2008), 117-136
Population P Systems with Moving Active Cells

Isidora Petreska\textsuperscript{1} and Petros Kefalas\textsuperscript{2}

\textsuperscript{1} South East European Research Centre (SEERC), 24 Proxenou Koromila Str., Thessaloniki 54622, Greece, ispetreska@seerc.org
\textsuperscript{2} CITY College, International Faculty of the University of Sheffield, 3 Leontos Sofou Str., Thessaloniki 54626, Greece, kefalas@city.academic.gr

Abstract. This work concentrates on the behaviour of biological multi-agent systems that consists of communities evolving in space and time, such as social insects, tissues, colonies of bacteria, etc. Such complex systems that are characterised with a highly dynamic structure can be modelled by utilizing population P systems with active cells. This class of P systems can be defined as an arbitrary graph-like membrane structure, that exhibits dynamic characteristics in terms of support to cell division, cell death, as well as reconfiguring of the whole system’s structure during its evolution. However, we argue that population P systems with active cells do not have consistent means and full support on the spatial characteristics found in nature, such as the position, the direction and the moving function of an individual. Therefore, we introduce a class of population P systems with moving active cells, namely $spPPS$. We argue that the spatial properties might lead to more accurate emergent behaviour on the macro-level of a multi-agent system. Moreover, we demonstrate how these properties might bring $spPPS$ closer to informal verification strategies and enhance their models with visual animation.

Keywords: formal modelling, membrane computing, population P systems, moving active cells

1 Introduction

There are many advances in the area of membrane computing up to date; and Population P systems (PPS) with active cells are considered to be one of them [1]. Defined as an arbitrary graph-like membrane structure, they exhibit rather unique characteristics towards modelling multi-agent systems (MAS) with a dynamic configuration (such as biological MAS) [1]. This work pays a particular interest in biological MAS comprised of communities evolving in space and time. Systems such as ant colonies, bacteria populations and skin-like tissues, are characterized with certain dynamicity found not only at the bonding concept, but also at their individual properties. A bonding concept refers to introducing new, or removing old communication links between cells/individuals, which in turn changes during the evolution of a system. On the other hand, individual
properties are formed by the notion that a cell can divide or get dissolved; individuals are being born or they die. These two aspects are actually the most prominent behaviour found in this rather complex application’s domain, completely supported by PPS with active cells through cell division, cell death and structure reconfiguration, accordingly. However, when it comes to evolution in space, whether it is at individual or community level, we argue that there is one more piece left out from the puzzle, i.e. the spatial characteristics found in nature. We argue that PPS with active cells do not have explicit means and full support to these characteristics, which could be simply defined as the position, the direction and the moving function of a cell or an individual.

This work implies the idea that the spatial properties might actually lead to more accurate emergent behaviour on the macro-level of a multi-agent system. Trying to use the existing PPS with active cells approach in such cases, would produce rather abstract models that are almost ineffective in practice. Therefore, we introduce another class PPS with moving active cells or spatial PPS ($^\text{sp}$PPS). Moreover, we demonstrate how these properties might bring $^\text{sp}$PPS closer to informal verification (such as simulation) and therefore enhance their models with visual animation.

Sect. 2 presents the advantages of PPS with active cells and provides a case study inspired from a simple ant colony. This leads to a discussion and discovery on some shortcoming PPS with active cells exhibit, in regards to the case study. The new $^\text{sp}$PPS formal definition is established in Sect. 3, supported by the benefits they offer towards spatial agent modelling. We demonstrate how $^\text{sp}$PPS would behave in practice, by modelling the aforesaid case study. Finally, the notion of verification and validation of $^\text{sp}$PPS is tackled in Sect. 4. We concentrate on an idea to developing a $^\text{sp}$PPS supporting tool towards informal verification, or more specifically visual animation. This work concludes with ideas for future research directions, as presented in Sect. 5.

2 Population P Systems with Active Cells: Discussion on Spatial Modelling

There are quite a few recent research approaches towards modelling spatial characteristics of biological systems. Some researchers apply process algebra to develop a calculus of processes that could describe the spatial geometric transformations (i.e. $\pi$-calculus [2] with geometric transformations), targeting applications in the area of cellular biology [3]. There were also recent attempts in PPS towards introducing geometric information, by defining membranes to represent spatial localisation [4]. Yet another agent-based approach is the intracellular NF-$\kappa$B signalling pathway for modelling spatial information in predictive complex biological systems [5]. A very recent work is also performed on modelling spatial characteristics of biological systems by utilizing the X-machines approach [6]. With this work however, we utilise the PPS with active cells approach. The idea behind is to maintain all the advantages of PPS with active cells, which makes them one of the most suitable formalisms for modelling the macro-level
of biologically inspired multi-agent systems with a highly dynamic nature. Some of the most prominent advantages of PPS with active cells are their support towards [7]:

- Non-deterministic communication – As opposed to deterministic communication, this property engages choices from indistinguishable possibilities, i.e., a communication rule is selected randomly;
- Dynamic addition and removal of agent instances – Cells can be introduced in (cell division) or removed from (cell death) the system, during the system’s evolution;
- Dynamic restructuring of the communications network – Cells can change the links (bonds) of their interconnection (form new bonds with other cells, or destroy existing bonds) as the system evolves. In other words, this implies to the concept of communication change;
- Maximal and arbitrary parallelism – These properties refer to the support of applying a maximal/arbitrary number of rules, but only one instance of any rule (selected non-deterministically) used at each computation step.

However, it was already introduced that PPS with active cells also exhibit certain shortcomings when it comes to spatial modelling. For the purpose of presenting them, let us consider a case study inspired from a simple ant colony, called Ant-Lines. Namely, we anticipate to model the behaviour of the ants within a colony, given that there is one leader ant moving towards a source of food, and every other ant is following the leader. At the beginning, all of the ants are in their nest. With the first evolution, the leader leaves the nest first, and chooses a random path to follow. Subsequently, one of the other ants leaves the nest as well, headed towards the current position of the leader. After a while, the second ant leaves the nest, and follows the ant ahead of it in the same manner. This process repeats until all of the ants reach the food source. It might be observed that this example looks like flocking.

This case study represents a phenomenon that could be modelled by employing the object-based parallel modelling power of PPS with active cells. The difficulties encountered in terms of modelling the spatial characteristics of the ants though, can be summarised as follows:

- There is a lack of supporting functions to describe movement. Functions, like movement to a random/specific position or heading to a random direction, are very commonly found in the biological systems in the nature and represent a common characteristic to every spatial system, as well.
- The position and the direction are not natively supported (we refer to predefined object types), and these properties are also common when it comes to spatial modelling.
- The concept of sensing can not be described with the existing communication rules. In other words, abstract information (objects that can not be quantified) are normally communicated by multiplication, i.e. perception which leaves multiple copies to both parties.
Furthermore, one might argue that having non-deterministically chosen cell evolution rules, happens in the biological systems and might represent one of the most important factors that leads to emergent behaviour. However, this is not true for all of the biological processes. Therefore, an interesting property that can be appended to this notion is prioritising of behaviours [8]. Referring to the case study, let us assume that there are some obstacles in the environment (for instance, water holes or fire places) and the ants are naturally (by instinct) constructed to avoid these obstacles. This would point to the fact that not all of the processes in nature are of equal importance, and there is a need to model some specific subset of rules in a way that they will be deterministically executed. Driven by the subsumption architecture [9], the work presented in [8] suggests defining a partial ordering over the evolution rules found in PPS with active cells.

3 Introduction to Spatial Population P Systems (spPPS)

The main contribution of this paper is introducing a variation of PPS with active cells, namely spPPS targeted to the spatial properties of a biological multi-agent system. This PPS class represents a continuation of an existing work towards prioritising the behaviours found in reactive, goal-oriented and hybrid intelligent agents, established in [8].

With the spPPS approach, the cells within the system, including the objects from the environment are represented such as $V = \{ w | w = (\alpha;v) \}$ for $\alpha \in Attributes$ (a set of attributes/labels), $v \in D$ ($D$ stands for domain and it is a set of values for $\alpha$).

The following spatial objects were introduced:
- $(\Pi_i;\pi_i) \in V$ is a predefined object denoting the cell’s position, with values $\pi_i \in \mathbb{N}_0 \times \mathbb{N}_0$ and a label $\Pi_i$ that stands for position, and
- $(\Delta_i;\delta_i) \in V$ is a predefined object denoting the cell’s direction, with values $0 \leq \delta_i \leq 360$ and a label $\Delta_i$ that stands for direction.

Formally, a cell can be defined as a tuple $C_i = (w_i, \Pi_i, \Delta_i, t_i)$, for each $1 \leq i \leq n$ (where $n$ is the total number of nodes/cells in the system), where $w_i \in V$ is a finite multiset of objects (such as $w_i = (\alpha_i;v_i)$), $t_i \in K$ is a type and the spatial properties (position and direction) are represented as $(\Pi_i;\pi_i)$ and $(\Delta_i;\delta_i)$.

Other changes that were introduced in the spPPS approach are the set of rules $R = R_e \cup R_s \cup R_{sp}$, where:
- $R_e$ is the finite set of cell evolution rules (communication, transformation, differentiation, division and death rules) found in the basic PPS with active cells;
- $R_s = \{(x:a); (y:b), in_copy)_t, ((x:a); (y:b), enter_copy)_t, ((y:b), exit_copy)_t\}$ is the finite set of sensing rules $(x:a) \in V, (y:b) \in V$, and
- $R_{sp}$ is a finite set of cell spatial rules.
The formal definition can be presented as follows: \( \text{spPPS} = (V, K, C_1, C_2, \ldots, C_n, w_E, \gamma, \alpha, R, O) \), such as:

- \( V \) is a collection of all the objects from all the cells within the system, including the objects from the environment;
- \( K \) is a collection of all the different types, associated with each individual cell in the system to identify different classes/types of cells;
- \( C_i = (w_i, \Pi_i, \Delta_i, t_i) \), for each \( 1 \leq i \leq n \) (where \( n \) is the total number of nodes/cells in the system);
- \( w_E \in V \) is the multiset of objects initially assigned to the environment;
- \( \gamma \) is the initial structure of the undirected graph, formally defined as:
  \[
  \gamma = \{(1, 2, \ldots, n), A\}, \text{ with } A \subseteq \{\{i, j\} \mid 1 \leq i \neq j \leq n\};
  \]
- \( \alpha \) is a finite set of bond making rules \((t, x_1; x_2, p)\), such as \( x_1, x_2 \in V \) and \( t, p \in K \);
- \( R = R_e \cup R_s \cup R_{sp} \);
- \( O \) is a partial order over the set of all evolution rules \( R \).

This definition is currently formulated to target spatial movement in 2-D space, however it can be easily generalised for spatial movement in an environment with any dimension n-D.

### 3.1 Elaboration

\( \text{spPPS} \) basically defined new types of predefined cell objects (that represent the cell’s position and direction), new type of communication rules (that define sensing properties) and spatial rules (used to establish movement functions). The detailed specifications are going to be further explained, as follows.

All the objects in the system, including the cell’s position and direction, as well as the objects that belong to the environment are represented in the form \((\text{label: value})\), where \text{label} is a descriptor of the object and \text{value} holds the actual object’s value, such as \((\text{age: N}_0)\) or \((\text{temperature: R})\). As for the cell’s position and direction, they are treated as all the rest of the objects, therefore the existing bond making rules found in the PPS with active cells definition, now allow manipulation with the spatial objects, i.e. \((t, (\Pi_1; \pi_1); (\Pi_2; \pi_2), p)\) where \( t, p \in K \), is a valid bond making rule. Similarly, the spatial objects can be used in all of the evolution rules as well, and there might be objects that belong to the environment of these types (\( \pi \) and \( \delta \)). \( \text{spPPS} \) however, define the following three new sensing communication rules: \((x:a); (y:b), \text{in}_\text{copy}t\), \((x:a); (y:b), \text{enter}_\text{copy}t\), and \((y:b), \text{exit}_\text{copy}t\). The difference from the existing communication rules is that now a copy of an object can be communicated through the cell’s membrane, rather than the actual object. If there is already an object within the cell with the same label as the copy, this object will be replaced with the new one. A graphical summary of all the evolution rules that exists in \( \text{spPPS} \) is provided in Fig. 1 (types of rules: i.–iii. communication, iv. transformation, v. differentiation, vi. division, vii. death rules and viii.–x. sensing communication rules).

In order to define movement and change of direction (to specific or random values), a particular type of rules, called spatial rules, were introduced as follows:
Fig. 1. Evolution rules in spPPS ($a \in V, b \in V$).

- $(A; \Pi, \text{move } B)_t$ where $B \in \mathbb{Z} \cup \{\text{Rand}\}$ – In presence of an object $A$ in a cell of type $t$, set $\Pi$ at position $B$ steps forward/backward (depending if $B$ holds a positive or negative value, respectively). If $B$ holds the value $\text{Rand}$, set $\Pi$ at a random number of steps forward/backward;

- $(A; \Pi, \text{set } B)_t$ where $Y \in \{(N_0 \times N_0) \cup \text{Rand}\}$ – In presence of an object $A$ in a cell of type $t$, set $\Pi$ at position $B$. If $B$ holds the value $\text{Rand}$, set $\Pi$ at a random position;

- $(A; \Delta, \text{set } B)_t$ where $B = \text{Rand}$ or $1 \leq B \leq 360$ – In presence of an object $A$ in a cell of type $t$, set $\Delta$ at direction $B$. If $B$ holds the value $\text{Rand}$, set $\Delta$ at a random direction;

And finally, for a given subset of evolution or spatial rules $r_1, r_2, r_3...$, the partial order $O$ would allow ordering, for example $r_1, r_2 \succ r_3$ means that any rule that among $r_1$ and $r_2$ can be non-deterministically chosen, as long as they execute before $r_3$.

### 3.2 Example: Ant-Lines

Referring again to the Ant-Lines case study, it can be modelled utilising the new spPPS approach on the following way (note that $x$ represents any):

- $V = \{(\text{nest\_position: } \pi), (\text{food\_position: } \pi), (\text{timer: } N_0), (\text{ant\_no: } N)\}$
- $K = \{\text{leader\_ant, follower\_ant}\}$
- $\gamma = \{\text{ant}_1, \text{ant}_2, \text{ant}_3..., \text{ant}_n\}$,
Population P Systems with Moving Active Cells

\[
\{\{\text{ant}_1, \text{ant}_2\}, \{\text{ant}_2, \text{ant}_3\}, \ldots \{\text{ant}_{n-1}, \text{ant}_n\}\}
\]

\[
\alpha = \{(\text{leader}_\text{ant}, (\text{ant}_\text{no}: 1); (\text{ant}_\text{no}: 2), \text{follower}_\text{ant})\},
\]

\[
\psi_E = \{(\text{nest}_\text{position}: (0, 0)), (\text{food}_\text{position}: (0, 50))\}
\]

\[
C_1 = \{(\text{II}_i: \text{nest}_\text{position}), (\Delta_i: 60), (\text{ant}_\text{no}: 1)\}, \text{leader}_\text{ant}
\]

\[
C_i = \{(\text{II}_i: \text{nest}_\text{position}), (\Delta_i: 60), (\text{ant}_\text{no}: i), (\text{timer}: 0)\}, \text{follower}_\text{ant},
\]

for \(2 \leq i \leq n\)

\[
R = \{r_1, r_2, \ldots, r_6\}, \text{such as:}
\]

\[
r_1 = ((\text{timer}: t) \rightarrow (\text{timer}: t + 1)) \text{follower}_\text{ant} - \text{transformation rule, for each}
\]

\[
r_2 = (\Pi_i) x \rightarrow \frac{x}{1}, \text{if } \Pi_i = \text{food}_\text{position}, \text{for } 1 \leq i \leq n - \text{death rule, once a cell}
\]

\[
r_3 = (x; \Delta_i, \text{set Rand}) \text{leader}_\text{ant} - \text{spatial rule, set random direction to a}
\]

\[
r_4 = (x; \Delta_i, \text{set } \Delta_i-1) \text{follower}_\text{ant} - \text{for } 2 \leq i \leq n - \text{spatial rule, set the direction of a cell}
\]

\[
r_5 = (x; \Pi_i, \text{move Rand}) \text{leader}_\text{ant}, \text{Rand} \in \{-1, 0, 1\} - \text{spatial rule, set}
\]

\[
r_6 = (\text{timer}; II_i, \text{move Rand}) \text{follower}_\text{ant}, \text{if } \text{timer} \geq 2 + i \text{ where } \text{Rand} \in \{-1, 0, 1\}, \text{for } 2 \leq i \leq n - \text{spatial rule, set random direction to a cell of a type}
\]

\[
O = \{r_1 \succ r_2 \succ r_3, r_4 \succ r_5, r_6\}
\]

The new definition of \textsuperscript{ap}PPS expanded the limitations and/or difficulties that were found in this modelling formalism when it comes to spatial agent properties, as discussed in Sect. 2. \textsuperscript{ap}PPS provide an intuitive and flexible way to model spatial and temporal properties. Moreover, it allows modelling of these characteristics with a great level of detail, which in turn would results to a more prominent emergent behaviour.

On each evolution cycle, all of the rules \(R\) are applied in parallel. The new spatial objects defined in \textsuperscript{ap}PPS are optional and applicable only to the cells that resemble biological moving individuals. On the other hand, an interesting concept introduced within the spatial rules is the awareness of randomness, a notion commonly found in the natural biological processes. Finally, the idea behind the new communication rules allows the sensing characteristics, or more particularly object multiplication.

One might argue that some of the improvements found in \textsuperscript{ap}PPS could be actually modelled with the standard PPS approach. However, by introducing all of the new spatial characteristics in the \textsuperscript{ap}PPS definition, we establish a greater idea than uniform representation, i.e. forming a basis towards informal validation of this modelling formalism. The details are presented in the following section.
The process that employs formal methods to confirm that a given model satisfies the system’s requirements, is referred to as a formal verification process [10]. One of the most widely used formal verification techniques is model checking, which focuses on thorough exploration on a predetermined state space, trying to conclude whether some properties of a complex system are being met. PPS are not supported with well founded formal verification techniques, however, this is not much in the interest of this work because spatial multi-agent systems are complex in nature. This means that even if there was a support to formal verification, utilizing PPS one would need an exponential time to complete the execution of a model checker, because a thorough exploration on the system’s state means all possible positions (coordinates) and directions.

Along these lines, PPS are not supported with testing strategies as well. Testing is a dynamic verification technique by providing all possible series of inputs and comparing its outputs with the documented specification [10]. Therefore, the same concept from this discussion (an exponential time to complete all the test) would apply for the idea of testing PPS models. This leaves simulation as the only appropriate way to confirm that the PPS model is having the intended behaviour. Simulation refers to executing scenarios (animations) and comparing the expected behaviour of the system to a textual or visual (visual animation) outcome. We claim that informal verification (or more specifically, simulation in a form of visual animation) in biologically inspired, spatial agent modelling provides the following benefits:

- **Benefit 1**: Detecting an emergent behaviour that cannot (or it is rather cumbersome to) be realised from a formal representation.
- **Benefit 2**: Visual ensurance that the behaviour of the system or an individual within the system, acts according to the intended specification/requirements.
- **Benefit 3**: Can act as a white box testing mechanism to systems that are NP-hard (their output can not be verified in polynomial time) or in some cases, even NP-complete.

These concepts are going to be supported by simulating the Ant Lines case study. For this purpose, NetLogo [11], [12] as a programmable platform for visual animation of multi-agent systems, is going to be utilised. The main reasons for choosing this software tool are: the support of large-scale systems, the utilisation of a functional language as a back-end (which in turn is perfect for representing an agent’s behaviour) and finally, the customisable graphical interface that facilitates modelling of the environment. The NetLogo simulation of this case study is presented in Fig. 2 (this simulation comes with the NetLogo’s built-in library).

For the purpose of investigating the trace of the ants’ path, Fig. 2 a) shows a mark being formed as the leader ant moves. Followed by the follower ants, it will be noticed that the shape of the ant line changes over time. Finally, a trace is left by the last ant, as shown in Fig. 2 b). Interesting observation at this point is the comparison of the two paths left from the initial and final ants; the path from the
leader is rather circuitous in comparison to the smooth shaped trail formed by the last ant. This simulation clearly demonstrates an emergent behavior that could not be detected from the dry PPS or $^s$PPS formal representation, i.e. Benefit 1. Furthermore, by observing the shape of the ant line as it is being formed, it can be ensured that the ants really follow the direction of the ant in front; which also ensures that the system acts according to the intended requirements as expected in Benefit 2. Finally, regarding the concept presented as Benefit 3, we bring into notice that NetLogo is a rather interactive environment, allowing for run-time user inputs and change of the global parameters.

It can be noticed that the simulation presented above is a manual translation of the $^s$PPS model. However, the new modifications introduced with this work allow a direct mapping between the $^s$PPS formal notation to a NetLogo code. For instance:

- (nest-x nest-y) and (food-x food-y) are the NetLogo definitions for the location of centre of nest and location of centre of food, supported in $^s$PPS by predefined object types (i.e. the position $\pi \in \mathbb{N}_0 \times \mathbb{N}_0$);
- leader-heading is representation of the heading of a leader ant, supported in $^s$PPS by the predefined object $\Delta$ denoting the cell’s direction;
- set leader-heading [heading] are net logo functions, now directly supported with the spatial rules; this specific function matches ($X; \Delta, \text{set } Y$);
- rt random-float angle is the NetLogo function that produces a random number (in this particular case, a floating point number) and assigns it to the direction of the ant. $^s$PPS can represent this behaviour by the notion of randomness introduced in the spatial rules;
- breed [leaders leader] and breed [followers follower] are supported by defining cell types.

There are many other similar examples that could demonstrate the logical link between NetLogo and $^s$PPS. This lead to the idea that by defining

![Fig. 2. NetLogo simulation of the Ant Lines example.](image-url)
some common rules for transformation between a $^s$PPS model and a NetLogo code, it is now possible to develop a tool that would do this translation automatically. This future direction will actually enhance $^s$PPS with informal verification strategies.

5 Conclusion and Future Work

We introduce a class of population P systems with moving active cells, namely $^s$PPS, which provide consistent means and full support on the spatial characteristics found in nature. Besides the theoretical contribution that was discussed, we conclude and focus on the practical benefits that were raised. Explicitly, there is a now a direct mapping between the $^s$PPS formal notation to a the NetLogo platform for visual simulation. Therefore, as a future research direction we are going to provide the rules for transformation between these two paradigms, and develop a tool for possibly automatic (or semi-automatic) translation of $^s$PPS models to executable NetLogo code.

There were quite a few other ideas for future work identified with this work, as well. One of them was the need to establishing another types of rules that deal with manipulation of the objects within the cells. This is very similar to what we did by introducing the sensing rules, but it can be further extended. Moreover, the concept of ordering rules raises a question towards how could dynamic ordering of rules be modelled. This means that at every cycle, the order by which the evolution rules execute should be recomputed. And finally, we tried to keep the spatial characteristics very simplistic by supporting the basic principles, but this notion can be extended much further.

Acknowledgements

The authors would like to thank Dr Marian Gheorghe for the useful discussion and his encouragement with his constructive comments on the first draft of this paper.

References


P systems simulating oracle computations

Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
{porreca, leporati, mauri, zandron}@disco.unimib.it

Abstract. We show how existing P systems with active membranes can be used as modules inside a larger P system; this allows us to simulate subroutines or oracles. As an application of this construction, which is (in principle) quite general, we provide a new, improved lower bound to the complexity class $\text{PMC}_{\text{AM}(-d,-\kappa)}$ of problems solved by polynomial-time P systems with (restricted) elementary active membranes: this class is proved to contain $\text{P}^{\text{PP}}$ and hence, by Toda’s theorem, the whole polynomial hierarchy.

1 Introduction

P systems with active membranes [5] are known to be able to solve problems beyond $\text{P}$ in polynomial time, by trading space for time. Using membrane division rules, an exponential number of membranes is created in polynomial time, which then compute in parallel, for instance exploring the solution space of an $\text{NP}$ problem.

The exact computing power of P systems with active membranes depends on which membrane division rules are available. First of all, it can be proved (Milano theorem [10]) that some form of division is actually necessary in order to go beyond $\text{P}$, otherwise the P system can be simulated sequentially in polynomial time. When the systems are allowed to divide membranes containing further membranes (called nonelementary membranes), thus replicating the whole structure in each copy of the dividing membrane, even $\text{PSPACE}$-complete problems become solvable in polynomial time. Usually, this involves constructing a membrane structure representing a full binary tree with exponentially many nodes [6,1].

When only membranes not containing further membranes (called elementary) can divide, the situation becomes more interesting: the systems are still able to create exponentially many membranes, but they all appear as leaves of the tree, without the possibility of creating more complicated tree structures. This is still sufficient not only to solve $\text{NP}$-complete problems [10], but also to count the number of “positive” candidate solutions in polynomial time; thus, the power of the whole complexity class $\text{PP}$ is captured [4].

In this paper we improve this lower bound by showing how P systems known to solve some problem (in polynomial time) can be used to simulate oracle queries, by embedding them into P systems simulating polynomial-time deterministic
Turing machines. The idea is to provide the input multiset to the embedded P systems not as part of their initial configuration, but only when the “outer” P systems require an oracle answer to continue its computation. Whenever this procedure can be carried out for a family of P systems deciding a language $L$, then the whole class $P^L$ of languages decidable in polynomial time by Turing machines with an oracle for $L$ can also be decided efficiently by P systems. We argue that this applies to most existing solutions described in the literature, though the formal details may vary.

As a concrete application, we choose $L$ to be the problem of checking whether the number of assignments satisfying a Boolean formula is greater than a given threshold. This problem is known to be PP-complete [4], and as a consequence the class $PP^P$ turns out to be solvable in polynomial time by P systems, without requiring nonelementary division or dissolution rules.

## 2 Definitions

P systems with active membranes are defined as follows.

**Definition 1.** A P system with active membranes of initial degree $d \geq 1$ is a tuple $Π = (Γ, Λ, μ, w_1, \ldots, w_d, R)$, where:

- $Γ$ is an alphabet, i.e., a finite non-empty set of symbols, usually called objects;
- $Λ$ is a finite set of labels for the membranes;
- $μ$ is a membrane structure (i.e., a rooted unordered tree, usually represented by nested brackets) consisting of $d$ membranes enumerated by $1, \ldots, d$; furthermore, each membrane is labeled by an element of $Λ$, not necessarily in a one-to-one way;
- $w_1, \ldots, w_d$ are strings over $Γ$, describing the initial multisets of objects placed in the $d$ regions of $μ$;
- $R$ is a finite set of rules.

Each membrane possesses, besides its label and position in $μ$, another attribute called electrical charge (or polarization), which can be either neutral (0), positive (+) or negative (−) and is always neutral before the beginning of the computation. The rules are of the following kinds:

- **Object evolution rules**, of the form $[a \to w]^α_h$. They can be applied inside a membrane labeled by $h$, having charge $α$ and containing an occurrence of the object $a$; the object $a$ is rewritten into the multiset $w$ (i.e., $a$ is removed from the multiset in $h$ and replaced by every object in $w$).
- **Send-in communication rules**, of the form $a [w]_h^α \to [b]_h^β$. They can be applied to a membrane labeled by $h$, having charge $α$ and such that the external region contains an occurrence of the object $a$; the object $a$ is sent into $h$ becoming $b$ and, simultaneously, the charge of $h$ is changed to $β$. 

---

434 Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, Claudio Zandron
Send-out communication rules, of the form \( [a]_h^\alpha \rightarrow [ ]_h^\beta b \)

They can be applied to a membrane labeled by \( h \), having charge \( \alpha \) and containing an occurrence of the object \( a \); the object \( a \) is sent out from \( h \) to the outside region becoming \( b \) and, simultaneously, the charge of \( h \) is changed to \( \beta \).

Dissolution rules, of the form \( [a]_h^\alpha \rightarrow b \)

They can be applied to a membrane labeled by \( h \), having charge \( \alpha \) and containing an occurrence of the object \( a \); the membrane \( h \) is dissolved and its contents are left in the surrounding region unaltered, except that an occurrence of \( a \) becomes \( b \).

Elementary division rules, of the form \( [a]_h^\alpha \rightarrow [ ]_h^\beta [ ]_h^\gamma \)

They can be applied to a membrane labeled by \( h \), having charge \( \alpha \), containing an occurrence of the object \( a \) but having no other membrane inside (an elementary membrane); the membrane is divided into two membranes having label \( h \) and charge \( \beta \) and \( \gamma \); the object \( a \) is replaced, respectively, by \( b \) and \( c \) while the other objects in the initial multiset are copied to both membranes.

Nonelementary division rules, of the form

\[
[1]_{h_1}^\delta \cdots [1]_{h_k}^\delta [1]_{h_{k+1}}^\epsilon \cdots [1]_{h_n}^\epsilon \rightarrow [1]_{h_1}^\beta \cdots [1]_{h_k}^\beta [1]_{h_{k+1}}^\gamma \cdots [1]_{h_n}^\gamma
\]

They can be applied to a membrane labeled by \( h \), having charge \( \alpha \), containing the positively charged membranes \( h_1, \ldots, h_k \), the negatively charged membranes \( h_{k+1}, \ldots, h_n \), and possibly some neutral membranes. The membrane \( h \) is divided into two copies having charge \( \beta \) and \( \gamma \), respectively; the positive children are placed inside the former, their charge changed to \( \delta \), while the negative ones are placed inside the latter, their charges changed to \( \epsilon \). Any neutral membrane inside \( h \) is duplicated and placed inside both copies.

Each instantaneous configuration of a P system with active membranes is described by the current membrane structure, including the electrical charges, together with the multisets located in the corresponding regions. A computation step changes the current configuration according to the following set of principles:

- Each object and membrane can be subject to at most one rule per step, except for object evolution rules (inside each membrane any number of evolution rules can be applied simultaneously).
- The application of rules is maximally parallel: each object appearing on the left-hand side of evolution, communication, dissolution or elementary division must be subject to exactly one of them (unless the current charge of the membrane prohibits it). The same reasoning applies to each membrane that can be involved to communication, dissolution, elementary or nonelementary division rules. In other words, the only objects and membranes that do not evolve are those associated with no rule, or only to rules that are not applicable due to the electrical charges.
- When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step.
While all the chosen rules are considered to be applied simultaneously during each computation step, they are logically applied in a bottom-up fashion: first, all evolution rules are applied to the elementary membranes, then all communication, dissolution and division rules; then the application proceeds towards the root of the membrane structure. In other words, each membrane evolves only after its internal configuration has been updated.

The outermost membrane cannot be divided or dissolved, and any object sent out from it cannot re-enter the system again.

The precise variant of P systems we use in this paper does not use dissolution or nonelementary division rules.

**Definition 2.** A P system with restricted elementary active membranes is a P system with active membranes where only object evolution, send-in, send-out, and elementary division rules are used.

A halting computation of the P system $\Pi$ is a finite sequence of configurations $C = (C_0, \ldots, C_k)$, where $C_0$ is the initial configuration, every $C_{i+1}$ is reachable by $C_i$ via a single computation step, and no rules can be applied anymore in $C_k$. A non-halting computation $C = (C_i : i \in \mathbb{N})$ consists of infinitely many configurations, again starting from the initial one and generated by successive computation steps, where the applicable rules are never exhausted.

P systems can be used as recognisers by employing two distinguished objects YES and NO; exactly one of these must be sent out from the outermost membrane during each computation, in order to signal acceptance or rejection respectively; we also assume that all computations are halting. If all computations starting from the same initial configuration are accepting, or all are rejecting, the P system is said to be confluent. If this is not necessarily the case, then we have a non-confluent P system, and the overall result is established as for nondeterministic Turing machines: it is acceptance iff an accepting computation exists. All P systems in this paper are confluent.

In order to solve decision problems (i.e., decide languages), we use families of recogniser P systems $\Pi = \{\Pi_x : x \in \Sigma^\star\}$. Each input $x$ is associated with a P system $\Pi_x$ that decides the membership of $x$ in the language $L \subseteq \Sigma^\star$ by accepting or rejecting. The mapping $x \mapsto \Pi_x$ must be efficiently computable for each input length [2].

**Definition 3.** A family of P systems $\Pi = \{\Pi_x : x \in \Sigma^\star\}$ is said to be (polynomial-time) uniform if the mapping $x \mapsto \Pi_x$ can be computed by two deterministic polynomial-time Turing machines $F$ (for “family”) and $E$ (for “encoding”) as follows:

- The machine $F$, taking as input the length $n$ of $x$ in unary notation, constructs a P system $\Pi_n$, which is common for all inputs of length $n$, with a distinguished input membrane.
- The machine $E$, on input $x$, outputs a multiset $w_x$ (an encoding of the specific input $x$).
Finally, $\Pi_x$ is simply $\Pi_n$ with $w_x$ added to the multiset placed inside its input membrane.\(^1\)

Any explicit encoding of $\Pi_x$ is allowed as output, as long as the number of membranes and objects represented by it does not exceed the length of the whole description, and the rules are listed one by one. This restriction is enforced in order to mimic a (hypothetical) realistic process of construction of the P systems, where membranes and objects are presumably placed in a constant amount during each construction step, and require actual physical space proportional to their number; see also [2] for further details on the encoding of P systems.

Finally, we describe how time complexity for families of recogniser P systems is measured, and their complexity classes [3].

**Definition 4.** A uniform family of P systems $\Pi = \{\Pi_x : x \in \Sigma^*\}$ is said to decide the language $L \subseteq \Sigma^*$ (in symbols $L(\Pi) = L$) in time $f : \mathbb{N} \to \mathbb{N}$ iff, for each $x \in \Sigma^*$,

- the system $\Pi_x$ accepts if $x \in L$, and rejects if $x \notin L$;
- each computation of $\Pi_x$ halts within $f(|x|)$ computation steps.

**Definition 5.** The complexity class $\text{PMC}_{\text{AM}(-d, -n)}$ consists of all the languages $L$ decidable in polynomial time by a uniform family of P systems with restricted elementary active membranes.

### 3 Simulating Turing machines

Let $M$ be a deterministic Turing machine working in polynomial time $p(n)$. We design a uniform family of recogniser P systems $\Pi_M$ simulating $M$ on inputs of different lengths. This construction is a variant of the one presented in [9].

A Turing machine $F_M$, on input $1^n$, produces as output a P system $\Pi_{M,n}$ having the following initial configuration:

$$[H [\ ] 0 [\ ] 1 [\ ] 2 \cdots [\ ] p(n)]_a^0.$$  

Each of the membranes having numerical label 0, 1, 2, $\ldots$, $p(n)$ represents one of the tape cells of $M$.\(^2\) The symbol contained in that cell is encoded by the charge of the associated membrane: a neutral charge represents a blank, a negative charge a 0, and a positive one a 1. In the following, we denote by lower case Greek letters $\alpha$, $\beta$ both the tape symbols and the corresponding charges.

During the first computation step, the object $H$ is replaced by $H_{0,0}^0$, using the following evolution rule:

$$[H \to H_{0,0}^0].$$  

\(^1\) Notice that this definition of uniformity is (possibly) weaker than the other one commonly used in membrane computing [3], where the Turing machine $F$ maps each input $x$ to a P system $\Pi_{s(x)}$, where $s : \Sigma^* \to \mathbb{N}$ is a measure of the size of the input; in our case, $s(x)$ is always $|x|$.

\(^2\) Recall that $M$ cannot visit more than $p(n) + 1$ tape cells on an input of length $n$. 


The reason for having a different object rather than directly \( h_0, q_0 \) in the initial configuration will be clarified below.) The object \( h_{i,q} \) simulates the tape head of \( M \): its subscripts denote the current position \( i \) on the tape, from 0 to \( p(n) \), and the current state \( q \) of \( M \). Their initial values are 0 and \( q_0 \), i.e., the leftmost position on the tape and the initial state of \( M \).

The transition function \( \delta : Q \times \Sigma \rightarrow Q \times I \times \{ \leftarrow, \rightarrow \} \) of \( M \) (where \( Q \) is the set of states, \( \Gamma \) the tape alphabet and \( \leftarrow, \rightarrow \) denote movement to the left and right respectively) is implemented by using communication rules, that are replicated for each possible membrane corresponding to a tape cell. A transition \( \delta(q_1, \alpha) = (q_2, \beta, \leftarrow) \) corresponds to the following rules:

\[
\begin{align*}
&h_{i,q_1} [^\alpha_i] \rightarrow [^\beta_{i-1}, q_2] \\
&[h_{i-1,q_2}]^\beta_i \rightarrow [^\beta_i h_{i-1,q_2}]
\end{align*}
\]

for \( 0 < i \leq p(n) \).

Notice how the first set of rules changes the charge of membrane \( i \) (representing the symbol in the \( i \)-th cell) and the two subscripts of \( h_{i,q} \) (representing head position and state) according to the transition function \( \delta \). The second set of rules is only used to move back the object \( h_{i,q_2} \) to its starting position, i.e., inside the outermost membrane \( s \). Transitions such as \( \delta(q_1, \alpha) = (q_2, \beta, \rightarrow) \), that move the head to the right, are analogous:

\[
\begin{align*}
&h_{i,q_1} [^\alpha_i] \rightarrow [^\beta_{i+1}, q_2] \\
&[h_{i+1,q_2}]^\beta_i \rightarrow [^\beta_i h_{i+1,q_2}]
\end{align*}
\]

for \( 0 \leq i < p(n) \).

Assuming without loss of generality that the transition function \( \delta \) is undefined on the accepting and rejecting states \( q_{\text{yes}} \) and \( q_{\text{no}} \), two further sets of rules are needed in order for \( \Pi_{M,n} \) to give the same result as \( M \) on its input:

\[
\begin{align*}
&[h_{i,q_{\text{yes}}}]^0 \rightarrow [^0_i \text{YES}] \\
&[h_{i,q_{\text{no}}}]^0 \rightarrow [^0_i \text{NO}]
\end{align*}
\]

for \( 0 \leq i \leq p(n) \).

The size of the membrane structure, the number of different objects and the rules needed to simulate \( M \) are bounded by \( O(p(n)) \). Both the membrane structure and the rules can be computed by the machine \( F \) in polynomial time with respect to that amount.

The input for \( \Pi_{M,n} \), encoding the actual string \( x \) on which \( M \) is to be run, is computed by another Turing machine \( E_M \) simply by subscripting the symbols in \( x \) by their position in the string. For instance,

\[
E_M(11001) = 1_0 1_1 0_2 0_3 1_4.
\]

This encoding can be clearly computed in polynomial time with respect to the length of \( x \).

The encoding multiset \( E_M(x) \) is placed inside membrane \( s \) of \( \Pi_{M,n} \), and it is used to initialise the charge of the first \( n \) membranes corresponding to tape
cells, according to the following rules:

\[
\begin{align*}
1_i [0] & \rightarrow [\#]^+ \text{ for } 0 \leq i < n, \\
0_i [0] & \rightarrow [\#]^- \text{ for } 0 \leq i < n,
\end{align*}
\]

where \(\#\) is a “junk” object that has no role for the rest of the computation. These rules are applied in parallel during the first computation step, and avoiding interference with them is the reason why the object \(h_{i,q}\) is also introduced only in that step, after which the real simulation begins.

The family \(\Pi_M\) is thus constructed by \(F_M\) and \(E_M\) in polynomial time, and simulates \(M\) on inputs of various lengths in linear time with respect to \(p(n)\).

4 Simulating oracle machines

We now show how polynomial-time Turing machines with an oracle for a language \(L\) can also be simulated, assuming that \(L\) itself can be recognised by a uniform family of P systems \(\Pi_L\). While the construction we are going to describe is, in principle, quite general, the technical details depend on the specific family \(\Pi_L\) and, in particular, on the encoding chosen for the instances of \(L\). Hence, as a concrete example, we assume that \(L\) is the problem \(\text{Threshold-3SAT}\), defined as follows.

**Problem 1 (Threshold-3SAT).** Given a 3CNF Boolean formula \(\varphi\) over \(m\) variables and a non-negative integer \(k < 2^m\), do more than \(k\) assignments (out of \(2^m\)) satisfy it?

This problem, which is \(\text{PP}\)-hard, has been recently proved [4] to be solvable in polynomial time by a uniform family of P systems with restricted elementary active membranes; as a consequence, the result \(\text{PP} \subseteq \text{PMC}_{\text{AM}(−d,−n)}\) holds. By selecting \(\text{Threshold-3SAT}\) as the oracle language we can prove a better lower bound to this complexity class.

**Theorem 1.** \(\text{P}^{\text{PP}} \subseteq \text{PMC}_{\text{AM}(−d,−n)}.\)

The instances of \(\text{Threshold-3SAT}\) are Boolean formulae in ternary conjunctive normal form, that is, conjunctions of clauses consisting of a disjunction of three literals (i.e., optionally negated variables) where each variable occurs at most once in each clause. There are \(8\binom{m}{3}\) possible clauses over \(m\) variables (there are \(\binom{m}{3}\) subsets of 3 variables out of \(m\), and \(2^3\) ways to negate them); hence a Boolean formula \(\varphi\) with that number of variables can be encoded as a binary string of length \(8\binom{m}{3}\), where the \(i\)-th bit is 1 if the \(i\)-th clause (under some fixed ordering) actually appears in \(\varphi\), and 0 otherwise. Since the integer \(k\) can be encoded using further \(m\) bits, the total length of the encoding of an instance of \(\text{Threshold-3SAT}\) is \(8\binom{m}{3} + m\).

Now let \(M\) be a polynomial-time Turing machine with access to an oracle for \(\text{Threshold-3SAT}\). We make a number of assumptions about the functioning of
M in order to simplify the simulation. (None of these assumptions causes a loss of generality, since we only care about polynomial running time and not about the precise polynomial.) First of all, we assume that M has a single tape, which is used as input space, scratch space, and as a place to write the oracle queries. The oracle querying procedure works as follows: first M writes down the query string y on the tape, then it moves the head to the first symbol of y, and finally enters the query state q?. In the next computation step, the state of M will be changed, in order to reflect the answer of the oracle to the question “is y ∈ L?” to q?, or qn, and the tape head will be repositioned to the first tape cell. A further assumption we are allowed to make is that all query strings have the same length ℓ = 8n + m, where ℓ is the largest integer of that form to be less than or equal to p(n); this is due to the fact that a pair (φ1, k1), where φ1 is a formula over m1 variables, is a positive instance of Threshold-3SAT if and only if (φ2, k2) is, where φ2 is a formula having the same clauses of φ1 but over m2 ≥ m1 variables (i.e., φ2 is a padded version of φ1) and k2 = 2m2−m1 × k1 (i.e., the number of required assignments is increased in order to reflect the fact that the “new” variables xm1+1, . . . , xm2 do not actually appear in φ2, hence their truth values do not change the overall evaluation of the formula). By choosing an appropriate encoding, each instance can be brought to length ℓ just by padding φ1 and k1 with enough zeroes; see the original paper [4] for the details.

Let Πℓ ∈ ΠL be the P system associated to the Threshold-3SAT instances of length ℓ = 8n + m. This P system has the following initial configuration [4]:

\[
\begin{bmatrix}
[1_{(0)}]_{m ℓ} [1_{0}]_{m 0} \cdots [1_{0}]_{0} O_{t+1} NO_{t+3} \end{bmatrix}_{N}
\]

where \( t = 4\ell - 3m + 4 \). The input for Πℓ is placed inside membrane IN, and it consists of an encoding of (φ, k) described as follows:

\[ E_L (φ, k) = \{ c_i : \text{the } i\text{-th clause does not appear in } φ, \text{ for } 1 \leq i \leq 8\binom{m}{3} \} \cup \{ k_i : \text{the } i\text{-th bit of } k \text{ (counting from 0)} \text{ is } 1, \text{ for } 0 \leq i \leq m - 1 \}. \]

We construct the P system ΠM,n simulating M on inputs of length n as follows:

\[
\begin{bmatrix}
H \mid \mid 1_{0} \mid \mid 1_{2} \cdots \mid \mid 2p(n) \mid \mid Π^0_{\ell} \mid \mid Π^0_{q_1} \mid \mid Π^0_{q_2} \cdots \mid \mid Π^0_{q_m} \mid \mid 1_{0} \end{bmatrix}
\]

This initial configuration contains p(n) copies of Πℓ, each one enclosed by a further membrane having label Q. The number p(n) is chosen as it is the maximum number of queries that M can perform. However, the initial configuration of the embedded P systems Πℓ is changed by erasing the initial objects and keeping only the membrane structure and the rules; this is required in order for these P systems to avoid starting their computation immediately, as their input will

\(^3\) Although, for the sake of a simpler exposition, this configuration contains several membranes having the same label, the labels can be made unique by subscribing them and replicating the rules accordingly (this does not affect the construction time by more than a polynomial amount).
be provided later during the computation of $\Pi_{M,n}$. Membrane $a$ will be used to store (in its charge) the result of an oracle query.

Notice how the number of simulated tape cells has been increased to $2p(n)+1$: this is due to the fact that we require all query strings to be of length $\ell$, hence we need to leave extra space on the tape for padding them to this length.

The simulation of $M$ by $\Pi_{M,n}$ works exactly as in Section 3 as long as $M$ does not enter its query state. We shall describe how oracle queries are simulated, first in an informal way, then by giving all the technical details.

### 4.1 Informal description of the simulation of oracle queries

This is an overview of the oracle query simulation:

1. The tape positions corresponding to the query string are inspected, and the multiset of objects $w$ encoding it is produced.
2. At the same time, one of unused copies of the embedded P systems $\Pi_\ell$ is chosen; it will be used to simulate the current query.
3. The objects in $w$ are moved to their initial position inside that copy of $\Pi_\ell$.
4. The objects missing from the initial configuration of $\Pi_\ell$ are created and moved to the correct membranes.
5. Now the embedded P system performs its computation as in [4], and produces the answer to the query.
6. Finally, the answer is communicated to the object simulating the tape head of $M$; it switches to the corresponding state and resumes simulating the Turing machine.

In more details, the procedure works as follows. When $M$ enters the query state $q$, the object $h_{i,q}$ is produced and moved to membrane $s$. According to the convention described above, the query string $y$ (necessarily of length $\ell$) is now located on tape cells $i, \ldots, i + \ell - 1$.

First, the object $h'_{i,q}$ is rewritten into the following multiset:

$$h'_{i,q} = C_{1,i} \cdot C'_{2,i+1} \cdot \cdots \cdot C'_{\ell-m,i+\ell-m-1} K'_{0,i+\ell-m} K'_{1,i+\ell-m+1} \cdots K'_{m-1,i+\ell-1}$$

The objects $C_{j,i}$ and $K_{j,i}$ represent potential input objects $c_j$ and $k_j$ for an instance of $\Pi_\ell$, which will be actually produced depending on the particular query string $y$. The procedure is the following one: each object $C_{j,i}$ and $K_{j,i}$ enters the corresponding membrane $j$, and is either rewritten into a "junk" object $\#$, or sent out as $c_j$ or $k_j$ depending on the symbol contained in the tape cell (thus simulating the encoding machine $E_L$ described above).

In the meantime, the object $h'_{i,q}$ nondeterministically selects one of the neutral membranes having label $q$ and "opens" it by setting its charge to positive. The P system $\Pi_\ell$ contained inside that membrane will be used to answer the current query. The object is moved back as $w_\ell$ to membrane $s$, where it "waits" for $\ell$ steps by decreasing its subscript.

While the object $W$ waits, the objects $C_j$ and $K_j$ that were actually produced (there are at most $\ell$ of them) move through the positive membrane $Q$ and inside
the membrane \( s \) of the selected copy of \( \Pi_\ell \), thus reaching their initial position. At that point, the subscript of \( w \) will have reached 0. The object \( w_0 \) then enters the active instance of \( \Pi_\ell \), and inside membrane \( s \) it produces by evolution the objects \( 1_{\ell-m+1} \), \( 0_{\ell+2} \), and \( NO_{\ell+4} \), and is itself rewritten into \( w'_0 \). Notice how the subscripts of 1, 0, and \( NO \) are incremented by one: this is done in order to give the opportunity to \( 1_{\ell-m+1} \) to move (as \( 1_{\ell-m} \)) to its initial position inside membrane \( e \), and also to \( w'_0 \) to exit \( \Pi_\ell \) and move back to membrane \( q \).

While \( w'_0 \) moves to membrane \( s \), the chosen embedded P system \( \Pi_\ell \) finally starts computing as if it were in stand-alone form, as in [4]. This computation requires a polynomial amount of time, after which either the object \( YES \) or \( NO \) will be sent out to the surrounding membrane \( q \). That result object then exits \( q \), setting its charge to negative (thus signalling that the enclosed P system \( \Pi_\ell \) has been used, and cannot be reused again) and moving to membrane \( \lambda \). When entering it, the charge is set to positive (if the result object is \( YES \)) or negative (if it is \( NO \)); the result object is simultaneously rewritten into \( \# \).

When membrane \( \lambda \) is non-neutral, the object \( w'_0 \) can enter it, and be sent out as \( h_{0,q} \), or \( h_{0,q} \), depending on the result; the charge of \( \lambda \) is also reset to neutral to allow further queries. The simulation now proceeds again as in Section 3, until another oracle query is made or until the computation of \( M \) finally terminates.

### 4.2 Technical details

When the simulated Turing machine enters the query state, an object \( H_{i,q} \) appears inside the outermost membrane \( s \) of \( \Pi_{M,n} \). It is immediately subject to the following evolution rule, which is replicated for \( 0 \leq i \leq 2p(n) \):

\[
[H_{i,q} \rightarrow H'_{i,q} \mid c_{1,i} \cdots c'_{\ell-m,i+\ell-m-1} K^0_{i,i+\ell-m} \cdots K'_{m-1,i+\ell-1}]^0
\]

During the next step, the objects \( c'_{j,i} \) enter the corresponding membrane labelled by \( i \), where they are either deleted (if that membrane represents a tape cell containing 1, i.e., if the \( j \)-th clause occurs in the input formula \( \varphi \)), or sent back out as \( c_j \) (if the tape cell contains 0).

\[
\begin{align*}
& [c'_{j,i}]^0 \rightarrow [c'_{j,i}]^0 \\
& [c'_{j,i}]^+ \rightarrow [c_j]^+ \\
& [c'_{j,i}]^- \rightarrow [c_j]^- \\
& \text{for } 0 \leq i \leq 2p(n) \text{ and } 1 \leq j \leq \ell - m \text{ and } \alpha \in \{+, -\}
\end{align*}
\]

The same occurs to the objects \( k_{j,i} \), except that they are deleted when the corresponding tape cell contains a 0, according to the encoding performed by the machine \( E_L \).

\[
\begin{align*}
& [k'_{j,i}]^0 \rightarrow [k'_{j,i}]^0 \\
& [k'_{j,i}]^+ \rightarrow [k_j]^+ \\
& [k'_{j,i}]^- \rightarrow [k_j]^- \\
& \text{for } 0 \leq i \leq 2p(n) \text{ and } 0 \leq j \leq m - 1 \text{ and } \alpha \in \{+, -\}
\end{align*}
\]
While the objects encoding the Threshold-3SAT instance are produced, the object $h'_{i,q}$ changes the charge of one of the neutral membranes labelled by $q$ to positive, and moves back to $s$ as $w_\ell$.

$$h'_{i,q} \left[ \right]_q^0 \rightarrow [H_{i,q}]_q^+$$
$$h'_{i,q} \left[ \right]_q^+ \rightarrow [w_\ell]_q^+$$

for $0 \leq i \leq 2p(n)$

The objects $c_j$ and $k_j$ (which are at most $\ell$ in number) are then sequentially moved to the system $\Pi_\ell$ corresponding to the positive membrane $q$.

$$c_j \left[ \right]_q^0 \rightarrow [c_j]_q^+$$
$$c_j \left[ \right]_q^+ \rightarrow [c_j]_q^0$$
$$k_j \left[ \right]_q^0 \rightarrow [k_j]_q^+$$
$$k_j \left[ \right]_q^+ \rightarrow [k_j]_q^0$$

for $1 \leq j \leq \ell - m$

The object $w_\ell$ “waits” until the last object has entered membrane $q$ by decreasing its subscript

$$[w_i \rightarrow w_{i-1}]_q^0$$

for $1 \leq i \leq \ell$

then it also enters the selected embedded $P$ system $\Pi_\ell$ in order to initialise its configuration:

$$w_0 \left[ \right]_q^+ \rightarrow [w_0]_q^+$$
$$w_0 \left[ \right]_q^0 \rightarrow [w_0]_q^0$$

$$[w_0 \rightarrow w'_0]_{\ell-m+1} [O_{\ell+2} NO_{\ell+4}]_q^0$$

The object $w'_0$ is sent back to membrane $s$, while the newly created objects reach their actual position and/or subscript inside $\Pi_\ell$, allowing the actual computation of the embedded $P$ system to start.

$$[w'_0]_{\ell+1} \rightarrow [w'_0]_{\ell+1}$$

$$[w'_0]_{\ell+2} \rightarrow [w'_0]_{\ell+2}$$

$$[w'_0]_{\ell+3} \rightarrow [w'_0]_{\ell+3}$$

When the computation of the active instance of $\Pi_\ell$ terminates, a result object is sent out to membrane $q$, and it is moved in order to set the charge of membrane $A$ appropriately.

$$[YES]_q^+ \rightarrow [YES]_q^-$$
$$[NO]_q^+ \rightarrow [NO]_q^-$$
$$YES \left[ \right]_A^{\#} \rightarrow \left[ \right]_A^{\#}$$
$$NO \left[ \right]_A^{\#} \rightarrow \left[ \right]_A^{\#}$$
Now the object \( w_0 \) can “read” the answer from the charge of \( a \), reset it to neutral and be rewritten into the corresponding object encoding the new state of the simulated machine \( M \).

\[
\begin{align*}
   w_0 \left[ \left| \right|_\alpha \right] & \rightarrow \left[ w_0 \right]_\alpha^\alpha \quad \text{for } \alpha \in \{+,-\} \\
   \left[ w_0 \right]_\alpha^+ & \rightarrow \left[ 0 \right]_\alpha \text{ h}_{0,q_1} \\
   \left[ w_0 \right]_\alpha^- & \rightarrow \left[ 0 \right]_\alpha \text{ h}_{0,q_2}
\end{align*}
\]

Now the simulation of \( M \) continues as in Section 3.

### 4.3 Main result

We are finally able to prove the result anticipated at the beginning of this section.

**Proof (Theorem 1).** The previous discussion shows how any polynomial-time Turing machine equipped with an oracle for \textsc{Threshold-3SAT} can be simulated with a polynomial slowdown. Since \textsc{Threshold-3SAT} is \textbf{PP}-hard, any other problem in \textbf{PP} can be efficiently reduced to it by the simulated Turing machine before performing the query. As a consequence, the whole class \( \textbf{P}^{\text{PP}} \) is included in \( \textbf{PMC}_{\text{AM}(−d,−n)} \).

By Toda’s theorem (\( \textbf{PH} \subseteq \textbf{P}^{\text{PP}} \)) [8], this result implies that the whole polynomial hierarchy \( \textbf{PH} \) is contained in \( \textbf{PMC}_{\text{AM}(−d,−n)} \), bringing this class closer to the known \( \textbf{PSPACE} \) upper bound [7].

### 5 Conclusions

We showed how the P systems of an existing uniform family \( \Pi \) with active membranes, deciding a language \( L \), can be embedded into other P systems simulating Turing machines in order to answer oracle queries for \( L \). Although the technical details of the construction depend on the specific family \( \Pi \) and on the encoding of the instances of the problem, we are confident that it can be adapted to most families of P systems with active membranes already described in the literature, and possibly to other variants of P systems. However, from a formal standpoint, this result is still to be proved.

By using this construction, we also improved the previously known lower bound to the complexity class \( \textbf{PMC}_{\text{AM}(−d,−n)} \) from \( \textbf{PP} \) to \( \textbf{P}^{\text{PP}} \); this also means that the polynomial hierarchy is contained in this class. Giving a more precise characterisation remains an open question; while at this point the equality \( \textbf{PMC}_{\text{AM}(−d,−n)} = \textbf{PSPACE} \) seems to be the most plausible outcome (that would imply that nonelementary membrane division rules are redundant), we think it is also worth investigating the possibility that the reverse inclusion \( \textbf{PMC}_{\text{AM}(−d,−n)} \subseteq \textbf{P}^{\text{PP}} \) also holds; this result would be much more interesting, as it doesn’t equate a complexity class for P systems with P or \( \textbf{PSPACE} \) as it usually happens.
Being able to generalise the oracle query simulation to arbitrary families would imply that whenever \( L \in \text{PMC}_{\text{AM}}(−d,−n) \), automatically the inclusion \( \text{P}^L \subseteq \text{PMC}_{\text{AM}}(−d,−n) \) also holds. Furthermore, if the P systems implementing the oracle could be modified in order to reset to their initial configuration after their computation terminates (a condition we conjecture to be true in the case of the family solving \textsc{Threshold-3SAT} [4]), a more elegant solution could be provided, where only a single embedded P system is needed, as it could be reused for subsequent queries.

References

Abstract. In this paper we introduce a new model of P systems that uses vectors of rules to describe a causal dependence relation between the executions of the rules. We also study their computational power by considering several restrictions on the types of the rules.

1 Introduction

Living organisms involve huge numbers of interacting substances that are the subjects of some chemical reactions. These organisms are usually divided into physically separated compartments which in general have some particular functional purposes. Bio-chemical reactions take place at the level of such compartments not only by creating, transforming, eliminating, transporting substances, but also by acting on the compartments themselves (creating and/or destroying compartments). Traditionally, while modeling such systems, the involved formal models usually have a big descriptional complexity because they aim to capture the execution of each process that might occur. One common feature of the proposed formal models was the fulfilling of the principle of causality: every event has a cause (for instance, the execution of some bio-chemical reaction triggers the execution of others). Often, this principle was “implemented” in the formal models by specifying that the resulted compounds which are produced by the execution of a rule, will trigger the execution of other rule(s). In this respect, the principle of causality exhibits a certain temporal order by which any later event is determined by the earlier one.

In a more general framework, one can establish a precise temporal order of the execution of the rules but not necessarily based on the resulted compounds of the execution of each individual rule. This might be the case when for instance one knows that two rules are executed in sequence (at two different moments) but do not know (or do not want to model) the underlying intermediate steps; hence, although the “output” of the execution of the first rule seems not to trigger the execution of the second rule, these rules are dependent.

In the present paper we propose a new computing model inspired by the above arguments and we investigate its computational power under some restrictions.

2 Preliminaries

In this section we briefly recall the basic notions and notations from the formal language theory that we will use along the present paper.
Given an alphabet \( V = \{a_1, \ldots, a_n\} \), the set of all strings over \( V \) is denoted by \( V^* \). The empty string is denoted by \( \lambda \). A language \( L \) is a subset of \( V^* \). The length of a string \( x \) is denoted by \( |x| \). If \( x \in V^* \) and \( a \in V \) then \( |x|_a \) represents the number of occurrences of \( a \) in \( x \). The length set of \( L \) is \( \{ |x| \mid x \in L \} \). If \( FL \) is a family of languages, then \( NFL \) denotes the family of length sets of languages in \( FL \). We denote by \( \text{REG} \), \( \text{CF} \), and \( \text{RE} \) the family of regular, context-free, and recursively enumerable languages, respectively. It is known (see [6]) that \( \text{NREG} = \text{NCF} \subset \text{NRE} \).

A multiset over \( V \) is a mapping \( G : V \to \mathbb{N} \), hence one can represent it as \( \{(a_1,M(a_1)), \ldots, (a_n,M(a_n))\} \). For our convenience, such multisets will be represented in our formalism by strings of symbols; for these strings the order of symbols does not matter, however the number of copies of an object in a given multiset equals the number of occurrences of the corresponding symbol in the associated string.

### 2.1 Lindenmayer Systems

A 0L system is a tuple \( H = (V,P,\omega) \), where \( V \) is a finite alphabet, \( P \) is a finite and complete set of productions \( P \subseteq V \times V^* \), and \( \omega \in V^* \) is the axiom. In a 0L system \( H = (V,P,\omega), x \in V^* \) directly derives \( y \in V^* \) (written as \( x \xrightarrow{\text{OL}}_H y \)) if \( x = x_1x_2 \ldots x_n, y = y_1y_2 \ldots y_n \), where \( x_i \in V, y_i \in V^* \), and \( x_i \rightarrow y_i \in P \), \( 1 \leq i \leq n \).

An ET0L system is a tuple \( H = (V,T,\omega,\Delta) \), where \( T = \{T_1, \ldots, T_k\} \) is a non-empty finite set of finite substitutions (tables) and \( H_i = (V,T_i,\omega), 1 \leq i \leq k \), is an 0L system; \( \Delta \subseteq V \), \( \Delta \neq \emptyset \) is the terminal alphabet. In an ET0L system \( H = (V,T,\omega,\Delta) \), \( x \) directly derives \( y \), with \( x, y \in V^* \) (written as \( x \xrightarrow{\text{ET0L}}_H y \)) if \( x \xrightarrow{\text{OL}}_{T_i} y \) for some \( 1 \leq i \leq k \). The transitive and reflexive closure of \( \xrightarrow{\text{ET0L}}_H \) is denoted by \( \xrightarrow{\ast}_H \). The language generated by an ET0L system \( H \) is \( L(H) = \{w \in \Delta^* \mid \omega \xrightarrow{\ast}_H w\} \).

It is known (see [6]) that ET0L systems using two tables generate the same family of languages as the one generated by arbitrary ET0L systems.

In addition, if we denote by \( \text{NET0L} \) the family of length sets of languages generated by ET0L systems, then it is known (see [5]) that \( \text{NCF} \subset \text{NET0L} \subset \text{NRE} \). For instance the set \( \{2^n \mid n \geq 0\} \in \text{NET0L} \setminus \text{NCF} \).

### 2.2 Register Machines

A register machine is a tuple \( M = (n,\mathcal{P},l_0,l_f) \), where \( n \) is the number of registers, \( \mathcal{P} \) is a finite set of uniquely labeled instructions (the program), \( l_0 \) is the initial label, \( l_f \) is the final label. The instructions are of the following types:

- \( l_1 : (\text{add}(r),l_2,l_3) \) — increment the contents of the register \( r \) and proceed non-deterministically to the instruction labeled with \( l_2 \) or \( l_3 \);
- \( l_2 : (\text{sub}(r),l_2,l_3) \) — if \( r \) is not empty then decrement the contents of the register \( r \) and proceed to the instruction labeled with \( l_2 \); otherwise proceed to
instruction labeled with $l_3$;
• $l_h$: halt – halts the machine.

A configuration of $M$ is a tuple $C = (l, r_1, \ldots, r_n)$ where $l$ is a label of an instruction and $r_i \in \mathbb{N}$, $1 \leq i \leq n$, is the content of register $i$. The initial configuration is $C_0 = (l_0, 0, \ldots, 0)$. A computation of $M$ is a (finite or infinite) sequence of configurations $C_0, C_1, C_2, \ldots$ where $C_0$ is the initial configuration and each $C = (l, r_1, \ldots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by applying the instruction labelled $l$ over the current contents of registers.

A register machine generates a set of natural numbers if it starts with all registers being empty and, after executing instructions from $\mathcal{P}$ (starting from instruction labeled with $l_0$), it halts ($M$ executes the instruction labeled with $l_h$); the result of such a halting computation is collected from register 1.

It is known (see [2]) that any recursively enumerable set of natural numbers can be generated by a register machine with at most 3 registers.

### 3 P Systems with Chained Rules

The reader is assumed to be aware of the basic models of P systems; we especially refer to [3] and [4] for more details regarding this topic.

A *P system with chained rules* (a PCR system, for short) of degree $m \geq 1$ is a construct

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0)$$

where

• $O$ is an alphabet of objects;
• $C \subseteq O$ is the set of catalysts;
• $\mu$ is a tree structure of $m \geq 1$ uniquely labelled membranes (which delimit the regions of $\Pi$); usually, the set of labels is $\{1, \ldots, m\}$;
• $w_i \in O^*$, for $1 \leq i \leq m$, are multisets of objects which are initially present in the regions of $\mu$ (as indicated by the index);
• $R_i$, $1 \leq i \leq m$, are finite sets of vectors of evolution rules. A vector of evolution rules is denoted as $(r_1, \ldots, r_k)$ and each $r_i$, $1 \leq i \leq k$, is a non-cooperative rule $a \rightarrow v$ or a catalytic rule $ca \rightarrow cv$, where $a \in O \setminus C$, $v \in ((O \setminus C) \times \{here, out, in\})^*$, and $c \in C$;
• $i_0 \in \{1, \ldots, m\}$ is the label of the output region of $\Pi$.

A configuration of $\Pi$ is a vector $(\alpha_1, \ldots, \alpha_m)$, where $\alpha_i \in O^*$, $1 \leq i \leq m$, is the multiset of objects present in the region $i$ of $\Pi$. The initial configuration of $\Pi$ is the vector $C_0 = (w_1, \ldots, w_m)$.

Let $R_i = \{v_{i(1,1)}, \ldots, v_{i(s_i,j)}\}$ where $1 \leq i \leq m$ and such that $s_i \geq 1$; in addition, let $v_{i,j} = (r_{i,j,1}, r_{i,j,2}, \ldots, r_{i,j,t_j})$, $1 \leq j \leq s_i$ and such that $t_j \geq 1$. In other words, the first index of a rule indicates the region where it belongs, the second index indicates the vector from which it belongs, and finally the last index indicates the position in the vector.
A computation of \( \Pi \) is a recursively defined sequence of configurations (possibly infinite)

\[ C_0, C_1, \ldots, C_k, C_{k+1}, \ldots \]

having the following properties:

- \( C_0 \) is the initial configuration;
- \( C_{k+1} = (w_{(k+1,1)}, \ldots, w_{(k+1,m)}) \) is obtained from \( C_k = (w_{(k,1)}, \ldots, w_{(k,m)}) \) by applying in a nondeterministic manner on each multiset \( w_{(k,i)} \), \( 1 \leq i \leq m \), a maximal multiset of rules (with competition on objects) of the following form (the multiset is expressed as a string indicating a union of multisets of rules belonging to \( v_{(i,j)} \), \( 1 \leq j \leq s_i \); also, the multiplicities of some rules might be 0):

\[
\text{a multiset of rules from } v_{(i,1)} \quad \text{a multiset of rules from } v_{(i,s_i)}
\]

\[
(P_{(i,1,1)} P_{(i,1,2)} \cdots P_{(i,1,t_1)}) \quad (P_{(i,s_i,1)} P_{(i,s_i,2)} \cdots P_{(i,s_i,t_{s_i})})
\]

In addition, this multiset of rules has to obey the following conditions: if \( p_{(i,s,t)} \geq 1 \), \( 1 \leq s \leq s_i \), \( 1 \leq t \leq t_{s_i} \) (i.e., the corresponding rule \( r_{(i,s,t)} \) will be applied) then it is obligatory that the rule \( r_{(i,s,t-f)} \) for \( 1 \leq f \leq t-1 \) was applied to the configuration \( C_{k-f} \) (providing that \( k-f \geq 0 \)). In other words, given the multiset \( w_{(k,i)} \) of the configuration \( C_k \), if a vector of chained rules \( v \) starts its application (that is, the first rule in the vector is applied), then in the next subsequent configurations the rest of the rules from \( v \) will be applied in order in consecutive steps (one rule by each subsequent configuration but such that every such rule can be applied several times). However, if a rule from a given position \( f \) in an already started vector of chained rules \( v \) cannot be applied (although it was supposed to be executed \(^1\)), then the execution of \( v \) is dropped (that is, for the current application of \( v \), the remaining rules are not executed anymore).

A computation of \( \Pi \) is a halting one if no rule can be applied (as described above) in the last configuration (the halting configuration). The result of a halting computation is the number of objects from \( O \) contained in the output region \( i_0 \), in the halting configuration. A non-halting computation yields no result. By collecting the results of all possible halting computations of a given P system \( \Pi \), one gets \( N(\Pi) \) – the set of all natural numbers generated by \( \Pi \).

The family of all sets of numbers computed by P systems with chained rules with at most \( m \) membranes and with a list of features \( f \) is denoted by \( NOPCRP_m(f) \). The features considered in this paper are \( \text{ncoo} \) (P systems using vectors composed only by non-cooperative rules) and \( \text{cat}_k \) (P systems using vectors composed by non-cooperative rules and catalytic rules with at most \( k \) catalysts).

\(^1\) This happens when either in the current multiset there is no object that triggers the execution of the rule, or it was a competition on the objects and this rule lost it.
The above definition can be relaxed such that in a halting configuration one counts only the symbols from a given alphabet \( \Sigma \subseteq O \).

**Example 1** The following P system with chained rules

\[
\Pi = (O, C, \mu, w_1, R_1, i_0)
\]

where

- \( O = \{a, b\} \);
- \( C = \emptyset \);
- \( \mu = \{1\} \);
- \( w_1 = ba \);
- \( R_1 = \{v_{(1,1)} : (b \rightarrow b, a \rightarrow a), v_{(1,2)} : (b \rightarrow \lambda)\} \);
- \( i_0 = 1 \).

generates the set \( \{2^n | n \geq 0\} \) (the well known non-semilinear set of natural numbers).

In the first step of computation, because of the nondeterminism, either the vector \( v_{(1,1)} \) or \( v_{(1,2)} \) (but not both) can be selected for application; this is because in the initial multiset there is only one object \( b \). Consequently, either the rule \( b \rightarrow b \) from \( v_{(1,1)} \) is applied (which will further permit the generation of more objects \( a \)) or \( b \rightarrow \lambda \) from \( v_{(1,2)} \) is applied (which stops the generation of objects \( a \) and halts the computation).

Next, if in the current configuration there exists the object \( b \) then, due to the nondeterministic selection of the rules to be applied, two computational branches can be followed:

- the rules \( a \rightarrow aa \) from \( v_{(1,1)} \) and \( b \rightarrow b \) from \( v_{(1,1)} \) are executed (hence the number of objects \( a \) is doubled and the computation continues);
- the rules \( a \rightarrow aa \) from \( v_{(1,1)} \) and \( b \rightarrow \lambda \) from \( v_{(1,2)} \) are executed (hence the number of objects \( a \) is doubled but the next configuration will be a halting one; in this case the output region contains \( a^{2n} \) objects, hence \( \Pi \) generates the set \( \{2^n | n \geq 0\} \)).

The computation of \( \Pi \) is described with more details in Table 1.
Table 1. The computation of $\Pi$. Because of the nondeterminism different vectors/rules can be applied to a given configuration yielding different next configurations. In order to distinguish them we placed at each such configuration a superscript index (moreover, the subscript index indicates the computational step). In addition, the rules selected for execution are emphasized.

It is known (see [4]) that for a large class of variants of P systems with symbol objects and multiset rewriting rules the number of membranes does not count in what concerns the computational power. In fact, via a simple encoding of the regions labels into objects (for example, the index associated to an object indicates the region were the corresponding object belongs) one can prove that any such a variant of P system with $m > 1$ membranes can be simulated by one having only one membrane (or two for catalytic P systems because the catalyst cannot be deleted, hence it always counts in the output multiset). Using a similar approach one can easily prove the following result.

**Theorem 1.** $NOPCRP_m(\text{ncoo}) = NOPCRP_1(\text{ncoo})$, for $m \geq 2$. 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$C_0 = ([1], ba)$</td>
<td>Nondeterministic choice 1: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_1^1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nondeterministic choice 2: $(b \rightarrow \lambda)$</td>
<td>$C_1^2$</td>
</tr>
<tr>
<td>1</td>
<td>$C_1^1 = ([1], ba)$</td>
<td>Nondeterministic choice 1: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_2^1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nondeterministic choice 2: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_2^2$</td>
</tr>
<tr>
<td></td>
<td>$C_1^2 = ([1], a^{2^0}) = ([1], a)$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nondeterministic choice 1: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_{k+1}^1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nondeterministic choice 2: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_{k+1}^2$</td>
</tr>
<tr>
<td></td>
<td>$C_2^2 = ([1], a^{2^{k-1}})$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$C_{k+1}^1 = ([1], ba^{2^k})$</td>
<td>Nondeterministic choice 1: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_{k+2}^1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nondeterministic choice 2: $(b \rightarrow b, a \rightarrow aa)$</td>
<td>$C_{k+2}^2$</td>
</tr>
<tr>
<td></td>
<td>$C_{k+2}^2 = ([1], a^{2^k})$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Here we will prove that P systems with chained rules using vectors composed only by non-cooperative rules can generate at least the same family of sets of numbers as \( \text{NET}0L \).

**Theorem 2.** \( \text{NOPCRP}_1(\text{ncoo}) \supseteq \text{NET}0L \).

*Proof.* We will simulate the computation of an arbitrary ET0L system \( H = (V, T, \omega, \Delta) \) with two tables \( (T = \{ T_1, T_2 \}) \) using a P system with chained rules \( \Pi = (O, C, \mu, w_1, R_1, i_0) \). Before defining each component of \( \Pi \) let us assume that \( V = \{ a_1, \ldots, a_n \}, \Delta = \{ a_1, \ldots, a_m \}, 1 \leq m \leq n \), and \( T_1 = \{ r_{(1,1)}, \ldots, r_{(1,s_1)} \} \), \( T_2 = \{ r_{(2,1)}, \ldots, r_{(2,s_2)} \} \). In addition, let \( \overline{V} = \{ \overline{a} \mid a \in V \} \) and the morphism \( h : V \to \overline{V} \) such that \( h(a_i) = \overline{a_i} \). Also, if \( r_{(i,j)} = a \to \alpha \in T_i, i \in \{ 1, 2 \}, 1 \leq j \leq s_i \), denote by \( r_{(i,j)} = a \to h(\alpha) \). Then, one can construct \( \Pi \) as follows:

\[
O = V \cup \overline{V} \cup \{ t, X, Y, v, \# \} \cup \{ t_{(i,j)} \mid i \in \{ 1, 2 \}, 1 \leq j \leq 2n + 2 \} \cup \{ v_i \mid 0 \leq i \leq m - 1 \};
\]
\[
C = \emptyset ;
\]
\[
\mu = [ ]_1 ;
\]
\[
w_1 = \omega t ;
\]
\[
R_1 = \{ (t \to t_{\text{card}(T_1)} x), (t \to t_{\text{card}(T_2)} x) \}
\]
\[
\cup \{ (t_{(i,0)} \to t_{(i,1)}, t_{(i,j \to \#)} \mid i \in \{ 1, 2 \}, 1 \leq j \leq s_i \}
\]
\[
\cup \{ (t_{(1,1)} \to t_{(1,2)}), (t_{(2,1)} \to t_{(2,2)}) \}
\]
\[
\cup \{ (t_{(i,j)} \to t_{(i,j+1)}, a_{(j-1)} \to \#) \mid i \in \{ 1, 2 \}, 2 \leq j \leq n + 1 \}
\]
\[
\cup \{ (\# \to \#) \}
\]
\[
\cup \{ (t_{(i,j)} \to t_{(i,j+1)}, h(a_{j-n-1}) \to a_{j-n-1}) \mid i \in \{ 1, 2 \}, n + 2 \leq j \leq 2n + 1 \}
\]
\[
\cup \{ (t_{(1,2n+2)} \to \lambda, X \to Y), (t_{(2,2n+2)} \to \lambda, X \to Y) \}
\]
\[
\cup \{ (Y \to t), (Y \to v_0) \}
\]
\[
\cup \{ (v_i \to v_{i+1}, a_{i+1} \to \#) \mid 0 \leq i \leq m - 2 \} \cup \{ (v_{m-1} \to \lambda, a_m \to \#) \};
\]
\[
i_0 = 1.
\]

The object \( t \) represents a “selector” – it will be used to nondeterministically choose which table of the ET0L system will be simulated. Because at the beginning of simulation the multiset \( w_1 \) contains exactly one object \( t \), then only one of the rules \( t \to t_{\text{card}(T_1)} x \) or \( t \to t_{\text{card}(T_2)} x \) is executed. In both cases, the object \( X \) will be used later (after the simulation of a table) to nondeterministically decide if a new application of a table has to be simulated, or if the current multiset corresponds to a word in \( L(H) \). For an object of type \( t_{(i,j)} \), the index \( i \in \{ 1, 2 \} \) denotes the table that is currently simulated and the index \( 1 \leq j \leq 2n + 2 \) signifies a change of state.

Let us assume that the rule \( t \to t_{\text{card}(T_1)} x \) is executed, hence the application of the \( T_1 \) is simulated by \( \Pi \). There are generated \( t_{\text{card}(T_1)} x \) objects which in the next computational step will trigger the execution of some (at least one and
possibly all) of the first rules in the vectors \( \{(t(1,0) \rightarrow t(1,1), r(1,j)) \mid 1 \leq j \leq s_1\} \). Hence, in the third step, some of the objects from \( V \) will be rewritten into their corresponding counterpart from \( \overline{V} \) (there are executed some rules of type \( \overline{t(1,j)} \), \( 1 \leq j \leq s_1 \) — these rules are associated with the rules from table \( T_1 \)). In addition the rule \( (t(1,1) \rightarrow t(1,2)) \) is executed. Two possible situations may occur:

- the current multiset contains objects from \( \overline{V} \), at least one object from \( V \), the objects \( X \) and (several copies of) \( t(1,2) \).
- the current multiset contains only objects from \( \overline{V} \), the objects \( X \) and (several copies of) \( t(1,2) \).

The first case illustrates the situation when not all the objects from \( V \) were rewritten, hence it represents a “wrong” simulation of an application of table 1. If this is the case, then those objects will be rewritten into \# (by the corresponding rules from the vectors \( (t(1,j) \rightarrow t(1,j+1), a(j-1) \rightarrow #) \), \( 2 \leq j \leq n+1 \)). If the symbol \# is generated then the computation will never stop because the rule \( # \rightarrow # \) will always be applied (no matter what objects will be produced); the computation yields no result.

The second case corresponds to a “correct” simulation of the maximal parallel use of productions in an ET0L system. In this case the rules of type \( a(j-1) \rightarrow # \) from the vectors \( (t(1,j) \rightarrow t(1,j+1), a(j-1) \rightarrow #) \), \( 2 \leq j \leq n+1 \), will never be executed, hence the multiset produced in the step \( n+3 \) will contain objects from \( \overline{V} \) and the objects \( t_{1,n+2} \) (in several copies) and \( X \). Next, in consecutive steps, the objects from \( \overline{V} \) will be rewritten back in their counterparts from \( V \) (by the rules from the vectors \( (t(1,j) \rightarrow t(1,j+1), h(a(j-n-1) \rightarrow a(j-n-1)) \), \( n+2 \leq j \leq 2n+1 \). After that all the resulting objects \( t(1,2n+2) \) are deleted at once (by the rule \( t(1,2n+2) \rightarrow \lambda \) from the vector \( (t(1,2n+2) \rightarrow \lambda, X \rightarrow Y) \) and this gives further the possibility for the rule \( X \rightarrow Y \) to be executed. At this point, the multiset contains objects from \( V \) and the object \( Y \). If \( Y \) will be rewritten into \( t \) (by the rule \( Y \rightarrow t \)), then the entire simulation is restarted. Otherwise, if \( Y \) will be rewritten into \( v \) (by the rule \( Y \rightarrow v \)), then \( H \) verifies that all the objects from \( V \) from the current multiset are also belonging to \( \Delta \). In case there exists an object that belongs to \( V \setminus \Delta \), then the trap symbol \# is generated again and the computation cycles forever, yielding no result.

Consequently, because we successfully simulated the computation of an arbitrary ET0L system with two tables, we proved that \( NOPCRP_{1(ncoo)} \supseteq NET0L \).

The following result shows the equality between the class of sets of numbers generated by \( P \) systems with chained rules when all the vectors contain either non-cooperative rules or catalytic rules (with only one catalyst) and the class of all recursively enumerable sets of natural numbers.

**Theorem 3.** \( NOPCRP_{2(cat_1)} = NRE \).

**Proof.** By invoking the Church-Turing thesis one can assume true the inclusion \( NOPCRP_{2(cat_1)} \subseteq NRE \). The opposite inclusion can be proved by simulating an \( n \)-register machine \( M = (n, P, l_0, l_b) \) with a PCR system we construct.
During the simulation of $M$, the current contents of register $1 \leq i \leq n$, will be represented by the multiplicity of object $a_i$.

Let $Π = (V, C, μ, w_1, w_2, R_1, R_2, i_0)$ a PCR system such that:

$$V = \{a_i \mid 1 \leq i \leq n\} \cup \{l_1 \mid l_1 : (add(r), l_2, l_3) \in \mathcal{P}\} \cup \{l_1, R_1, R_2, R_{i_1}, R_0, \overline{R_0} \mid l_1 : (sub(r), l_2, l_3) \in \mathcal{P}\} \cup \{c\};$$

$$C = \{c\};$$

$$μ = \left[\begin{array}{l}1 \\ 2\end{array}\right];$$

$$w_1 = \emptyset;$$

$$w_2 = cl_0;$$

$$R_1 = \emptyset;$$

$$i_0 = 1;$$

and $R_2$ is defined as follows:

- for each instruction $l_1 : (add(r), l_2, l_3) \in \mathcal{P}$ we add to $R_2$ the rules:

  $$(l_1 \rightarrow a_i, l_2)$$

  $$(l_1 \rightarrow a_i, l_3)$$

- for each instruction $l_1 : (sub(r), l_2, l_3) \in \mathcal{P}$ we add to $R_2$ the rules:

  $$(l_1 \rightarrow l_i R_{l_i}, c a_r \rightarrow c, l_i \rightarrow l_2, l_2 \rightarrow l_2),$$

  $$(R_{l_i} \rightarrow \overline{R_{l_i}}, \overline{R_{l_i}} \rightarrow \lambda, l_i \rightarrow l_3).$$

- for each instruction $l_h : (halt) \in \mathcal{P}$ we add to $R_2$ the rules:

  $$(l_h \rightarrow λ, a_1 \rightarrow a_{1, out})$$

The system $Π$ works as follows. Initially $Π$ starts its computation having in region 2 the multiset $cl_0$ (this corresponds to the starting configuration of $M$ when all its registers are empty and the label of the rule to be executed is $l_0$). $Π$ performs its computation by simulating the execution of instructions from the $\mathcal{P}$.

Now, let us assume that, at a certain moment during the computation of $M$, the current configuration of $M$ is $l_1, r_1, \ldots, r_n$, where $r_i \in \mathbb{N}$, $1 \leq i \leq n$; in addition, let us suppose the instruction to be executed by $M$ is $l_1 : (add(r), l_2, l_3) \in \mathcal{P}$. Correspondingly, $Π$ will have in region 2 the multiset $a^r_1 \ldots a^n r cl_3$. Since there is only one object $l_1$, the rule that can be executed by $Π$ is either $(l_1 \rightarrow a_i, l_2)$ or $(l_1 \rightarrow a_i, l_3)$; nondeterministically only one of them will be actually executed yielding the multiset $a^r_1 \ldots a^n r cl_2$ or $a^r_1 \ldots a^n r + 1 \ldots a^n r cl_3$, respectively. These multisets correspond to one of the next possible configurations of $M$, i.e., $(l_2, r_1, \ldots, r + 1, \ldots, r_n)$ or $(l_3, r_1, \ldots, r + 1, \ldots, r_n)$. Consequently, the incrementing instruction of $M$ was successfully simulated by $Π$. 

*P Systems with Chained Rules* 455
In case the instruction to be executed by $M$ is $l_1 : (\text{sub}(r), l_2, l_3) \in \mathcal{P}$ then $\Pi$ will proceed with the simulation as follows. The rule $l_1 \rightarrow \overline{l}_1 R_{l_1}$ will be executed in the first step of simulation. In the second step of simulation we distinguish two possible cases: either $r = 0$ (that is, object $a_r$ is missing from region 2) or $r > 0$.

If $r = 0$ the only rule that can be executed is $R_{l_1} \rightarrow \lambda$. In addition, the execution of the vector $(l_1 \rightarrow \overline{l}_1 R_{l_1}, c a_r \rightarrow c, \overline{l}_1 \rightarrow l_2, \overline{l}_2 \rightarrow l_2)$ is interrupted because there is no object $a_r$. Hence in the subsequent steps the rules $R_{l_1} \rightarrow \lambda$ and $\overline{l}_1 \rightarrow l_3$ will be executed (in order, in consecutive configurations). Consequently, in the fourth step of the simulation the multiset in region 2 of $\Pi$ will be $a_1^{r_1}, \ldots, a_n^{r_n} c l_3$ which corresponds to the configuration $(l_3, r_1, \ldots, r = 0, \ldots, r_n)$ of $M$.

If $r > 0$ the rules that will be simultaneously executed are $c a_r \rightarrow c$ and $R_{l_1} \rightarrow \lambda$. In the next computational step there will be simultaneously executed the rules $\overline{l}_1 \rightarrow l_2$ and $\overline{l}_2 \rightarrow \lambda$. Finally, in the fourth step of the simulation the only rule that will be executed is $l_2 \rightarrow l_2$, hence the multiset in region 2 of $\Pi$ will be $a_1^{r_1}, \ldots, a_n^{r_n} c l_2$ which corresponds to the configuration $(l_2, r_1, \ldots, r = 0, \ldots, r_n)$ of $M$.

In case $l_h$ is generated, then all the objects $a_1$ are sent out the output region 1 (by the rule $a_1 \rightarrow a_{1,\text{out}}$) and the computation stops. It follows that the entire program $\mathcal{P}$ can be successfully simulated by $\Pi$, hence $\Pi$ generates the same set of numbers as $M$.

In conclusion, we have shown that $NRE \subseteq \text{NOPCRP}_2(\text{cat}_1)$. Consequently, we have proved that $\text{NOPCRP}_2(\text{cat}_1) = NRE$.

4 Conclusions

A necessary condition (but not sufficient) in the principle of causality is the delay of one event relatively to another. The sufficient condition imposes the precise definition of what event is the cause and what event is the effect. Traditionally, the principle of causality was implicitly modeled (and/or understood) in the P system framework by specifying that the objects resulted after applying a rule constitute the cause that might trigger the application(s) of other rules. Starting from this general feature several variants of P systems were proposed in order to refine the original definition and to better express the control flow imposed by this principle. In this respect were proposed P systems with weak/strong priorities, P systems with promoters/inhibitors, and so on. In this paper we introduced a new computational model by specifying a certain causal dependence relation between the execution of the rules.

There are several differences between the presented model and P systems with promoters. According with their definition promoters act globally and even one promoter might trigger several rules. For example, if one considers the multiset $a^m b^n p$ with $m, n \geq 1$ and the rules

$$r_1 : a \rightarrow \alpha|_p$$
$$r_2 : b \rightarrow \beta|_p$$
then both rules will be applied. In the proposed model, it might seem that in a vector \((p \rightarrow p, a \rightarrow \alpha)\) the execution of the rule \(p \rightarrow p\) might “activate” in the next computation step the rule \(a \rightarrow \alpha\) (hence it seems that the promoted rule \(r_1 : a \rightarrow \alpha|_p\) can be easily simulated). However, this is not the case when one has to simulate the execution of a couple of rules (as \(r_1\) and \(r_2\)), because simply considering in addition the vector \((p \rightarrow p, b \rightarrow \beta)\) will not solve the problem (recall that there is only one object \(p\), hence either the rule \(a \rightarrow \alpha\) or \(b \rightarrow \beta\) will be executed in the second step). Similar situations can be imagined while comparing the proposed model with P systems with weak/strong priorities.

Several open problems and research lines can be formulated for the model we introduced. For example, it is not known the exact relation between the computational power of this model and P systems with promoters/priorities (for this problem we conjecture that they have equal computational power). A novel research line aim to use for example a more general structure to describe the order by which the rules are applied.

Acknowledgements

The work of the author was supported by the CNCSIS IDEI-PCE grant, no. 551/2009, Romanian Ministry of Education, Research and Innovation.

References

Modelling, testing and verification of P systems with active membranes using Rodin and ProB*

Adrian Țurcanu, Florentin Ipate
Department of Computer Science, University of Pitesti

1 Introduction

Membrane computing, the field initiated by Gheorghe Păun [17], studies computing devices, called P systems, inspired by the functioning and structure of the living cell. In the last years, the research on various programming approaches related to P systems ([6], [19]) and formal semantics ([4], [2], [15]), or with respect to decidability of some model checking properties [5], has created the need for methods for formally verifying and testing such systems.

Formal verification has been studied for different variants of P systems by using rewriting logic and the Maude tool [2] or, for stochastic systems [3], PRISM and the associated probabilistic temporal logic [9]. More recently, NuSMV [8], SPIN [14] and Rodin [10], [11] have been used to verify various properties of transition P systems. Various approaches to build test cases for such P systems have also been proposed [7], [12], [13].

Event-B is a formal modelling language introduced about 10 years ago by J.R. Abrial [1], used for developing mathematical models of complex systems which behave in a discrete fashion. Event-B is an evolution of the B language, one of the most used modelling language in industry since its introduction in the 90s. The efforts for developing Event-B have been supported by two European research projects: RODIN\(^1\), which produced a first platform for Event-B called Rodin, and DEPLOY\(^2\), which is currently enhancing this platform based on feedback from its industrial partners (Bosch, SAP, Siemens and Space Systems Finland), which experiment with the latest development of the platform. The core technology behind Rodin platform is theorem-proving, but also model-checking (ProB) or animation tools (Anim-B) have been integrated as plug-ins.

In this paper we propose an approach for modelling and verifying P systems with active membranes, based on Event-B and its associated model-checker, ProB. Given the industrial support for Event-B and the strength of the Rodin

\(^*\)ACKNOWLEDGMENT: This work was partially supported by the strategic grant POSDRU 88/1.5/S/52826, Project ID52826 (2009), co-financed by the European Social Fund - Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013 for Adrian Țurcanu, and by project Deploy, EC-grant no. 214158, and Romanian Research Grant CNCSIS - UEFISCUS, project number PNII - IDEI 643/2008 for Florentin Ipate.

\(^1\)http://rodin.cs.ncl.ac.uk - Project running between 2004-2007

\(^2\)http://deploy-project.eu - Project running between 2008-2012
platform, we believe that this approach will have an important impact on the practical use of P systems.

2 Background

2.1 P systems

P systems are distributed and parallel models processing multisets of objects using evolution, communication and other types of rules encapsulated into regions delimited by membranes.

In a P system with active membranes not only the objects evolve but also the membrane structure.

Definition. A P system with active membranes is a tuple \( \Pi = (V, \mu, w_1, ..., w_n, R_1, ..., R_n) \), where \( V \) is a finite set of objects, called alphabet; \( \mu \) defines the initial membrane structure, which is a hierarchical arrangement of \( n \) compartments called regions delimited by membranes - these membranes and regions are identified by integers 1 to \( n \); \( w_i, 1 \leq i \leq n \), represents the initial multiset occurring in region \( i \); \( R_i, 1 \leq i \leq n \), denotes the set of processing rules applied in region \( i \).

Electrical charges from the set \( \{+, -, 0\} \) can also be associated with membranes, obtaining P systems with polarizations.

The basic types of rules associated with active membranes are [18]:

- evolution rules, associated with membranes and depending on the label and the charge of the membrane:
  \( [u \rightarrow v]^p_i \), \( p \in \{+, -, 0\}, u, v \in V^* \);
- communication rules, sending an object into a membrane and possibly changing its polarization:
  \( u[\_]^p_i \rightarrow [v]^p_i \), \( p, fp \in \{+, -, 0\}, u, v \in V \);
- communication rules, sending an object out of a membrane and possibly changing the polarization of the membrane:
  \( [u]^p_i \rightarrow [\_]^{fp_i} \), \( p, fp \in \{+, -, 0\}, u, v \in V \);
- dissolving rules (inapplicable to skin):
  \( [u]^p_i \rightarrow v, p \in \{+, -, 0\}, u, v \in V \);
- division rules (inapplicable to skin):
  \( [u]^p_i \rightarrow [v]^{fp_1} [w]^{fp_2} \), \( p, fp_1, fp_2 \in \{+, -, 0\}, u, v, w \in V \) (all objects different from \( u \) are duplicated in the two new membranes).

A configuration of the P system \( \Pi \), contains the current membrane structure and the multisets associated with each region; the initial configuration is \((\mu, w_1, ..., w_n)\).
2.2 Event-B, Rodin and ProB

The Event-B models are abstract state machines in which transitions between states are implemented as events. An Event-B model is made of several components. Each component can be either a machine or a context. Contexts contain the static structure of the system: sets, constants and axioms. Axioms define the main properties of sets and constants. On the other hand, machines contain the dynamic structure of the system: variables, invariants, and events. Invariants state the properties of variables and events define the dynamic of the transition system.

An event is a state transition which is specified in terms of guards and actions. The guards are necessary conditions for an event to be enabled. They are theorems derivable from invariants, axioms and previously declared guards. An event may have no guards; in this case it is permanently enabled. The actions describe how the occurrence of an event will modify some of the variables of the machine. All actions of an event are performed at the same time. An action might be either deterministic (using the normal assignment operator “:=”) or non-deterministic. There are two forms of non-deterministic actions: one using the “∈” operator and the other using the operator “|”. In the first case, the action “x ∈ \{ set of possible values \}” has the following result: an arbitrarily chosen value from the set of possible values is assigned to the variable x. In the second case, the expression “x | P(x)” means that the variable x receives a value such that the predicate P is true.

A very important Event-B concept is refinement, which allows a model to be developed gradually.

When the model is finished, a Rodin Platform tool, called Proof Obligation Generator, decides what is to be proved in order to ensure the correctness of the model (e.g. invariant preservation, consistency between original and refined models). Therefore, the proving mechanism provides the guarantee of a formally correct model before the model checker is actually used. This is a big strength of the Rodin platform.

ProB is an animation and model checking tool which accepts B-models, but is also integrated within the Rodin platform. Unlike, most model checking tools, ProB works on higher-level formalisms and so it enables a more convenient modelling.

Properties of an Event-B model can be verified using either the ProB version within the Rodin platform or the standalone version, which offers a greater range of facilities, such as computation of operation coverage or the possibility to find states satisfying a predicate or enabling an operation. When the standalone version is used, the model can be automatically translated into the B language and imported into ProB.

ProB supports automated consistency checking, which can be used to detect various errors in B specifications. The animation facilities allow: to visualize, at any moment, the state space, to execute a given number of operations, to see the shortest trace to current state. Properties that are intended to be verified can be formulated using the LTL or the CTL formalism.
3 The Event-B model of P systems with active membranes

In this section we present the main ideas about how to build the Event-B model of a P system with active membranes.

We consider a P system with active membranes and polarizations over the alphabet \( V = \{a, b, c\} \), with two membranes initially.

The first component of the model is a context that contains the set of symbols (denoted SYMBOLS), the set of electrical charges (denoted CHARGES) (all their values must be distinct) and the axioms that state their properties.

The second component is a machine with the following variables:

- two partial functions used to represent the structure of the P system, except the skin: \( cell, cellp : \mathbb{N} \rightarrow (\text{SYMBOLS} \rightarrow \mathbb{N}) \) with the following signification: \( cell(x)(a) \) represents the number of objects \( a \) in the membrane \( x \), and \( cellp(x)(a) \) represent the number of objects \( a \) produced in the membrane \( x \) between two steps of maximal parallelism;

- two functions used to represent the object existing and respectively produced between two steps of maximal parallelism in the skin: \( skin, skinp : \text{SYMBOLS} \rightarrow \mathbb{N} \);

- two partial functions \( charge, fcharge : \mathbb{N} \rightarrow \text{CHARGES} \), where \( charge(x) \) and, respectively \( fcharge(x) \) represent the initial, respectively the final charge of membrane \( x \). We use these to satisfy the requirement that, at each step of maximal parallelism, only rules that have the same initial and final charge can be applied. Besides the usual charge values (plus, minus and zero), we introduce an intermediate value, \( all \), that we use as an initial value for each \( fcharge(x) \).

- \( skinc \) and \( fskinc \) representing the initial, respectively the final charge of the skin;

- \( mark : \mathbb{N} \rightarrow \{0, 1\} \) a partial function used “to mark” cells produced in a division rule or involved in a communication rule between two steps of maximal parallelism; such cells cannot be the subject of another division or communication rule in the same computation step.

For each rule we introduce an event. The event is enabled if the rule can be applied and its application modify the state of the system accordingly. We consider division, evolution and communication rules. Since the applications presented in this paper do not involve dissolution rules, for simplicity, such rules are not considered in this section. However, the model presented here can be extended in a straightforward manner to include such rules.

For example, the event corresponding to the division rule \( [a]_2^0 \rightarrow [b]_2^0 [c]_2^+ \) will have the following structure:

**DivisionRule**

\[
\begin{align*}
\text{any} \\
\quad x \\
\quad y
\end{align*}
\]
where

\begin{align*}
  \text{grd1} : & \; x \in \text{dom}(\text{cell}) \\
  \text{grd2} : & \; y \in \mathbb{N} \setminus \text{dom}(\text{cell}) \\
  \text{grd3} : & \; z \in \mathbb{N} \setminus \text{dom}(\text{cell}) \\
  \text{grd4} : & \; \text{cell}(x)(a) \geq 1 \\
  \text{grd5} : & \; \text{charge}(x) = \text{zero} \\
  \text{grd6} : & \; \text{mark}(x) = 0 \\
  \text{grd7} : & \; y \neq z
\end{align*}

then

\begin{align*}
  \text{act1} : & \; \text{cell} := \{ y \mapsto \{ a \mapsto (\text{cell}(x)(a) + \text{cellp}(x)(a) - 1), \\
                  & \quad b \mapsto (\text{cell}(x)(b) + \text{cellp}(x)(b) + 1), \\
                  & \quad c \mapsto (\text{cell}(x)(c) + \text{cellp}(x)(c)), \\
                  & \quad z \mapsto \{ a \mapsto (\text{cell}(x)(a) + \text{cellp}(x)(a) - 1), \\
                  & \quad b \mapsto (\text{cell}(x)(b) + \text{cellp}(x)(b)), \\
                  & \quad c \mapsto (\text{cell}(x)(c) + \text{cellp}(x)(c) + 1) \} \cup \{ x \} \triangleleft \text{cell} \} \\
  \text{act2} : & \; \text{cellp} := \{ \{ x \} \triangleleft \text{cellp} \} \cup \{ y \mapsto \{ a \mapsto 0, b \mapsto 0, c \mapsto 0 \}, \\
                  & \quad z \mapsto \{ a \mapsto 0, b \mapsto 0, c \mapsto 0 \} \} \\
  \text{act3} : & \; \text{charge} := \{ \{ x \} \triangleleft \text{charge} \} \cup \{ y \mapsto \text{zero}, z \mapsto \text{plus} \} \\
  \text{act4} : & \; \text{mark} := \{ \{ x \} \triangleleft \text{mark} \} \cup \{ y \mapsto 1, z \mapsto 1 \} \\
  \text{act5} : & \; \text{fcharge} := \{ \{ x \} \triangleleft \text{fcharge} \} \cup \{ y \mapsto \text{all}, z \mapsto \text{all} \}
\end{align*}

end

In this case, any cell \( x \) with the charge \( 0 \) (grd5), that contains at least one object \( a \) (grd4) and unmarked (grd6) is divided into two cells \( y \) and \( z \). Cell \( x \) is removed from the domain of the partial functions \( \text{cell}, \text{cellp}, \text{charge}, \text{fcharge} \) and \( \text{mark} \) (using the domain substraction operator \( \triangleleft \)), and cells \( y \) and \( z \) are added to these together with the corresponding objects. Both new cells are marked (act4).

For the evolution rule \( [b \rightarrow a]^0_2 \), the corresponding event is:

**EvolutionRule**

any

\( x \)

where

\begin{align*}
  \text{grd1} : & \; x \in \text{dom}(\text{cell}) \\
  \text{grd2} : & \; \text{cell}(x)(b) \geq 1 \\
  \text{grd3} : & \; \text{charge}(x) = \text{zero} \\
  \text{grd4} : & \; \text{mark}(x) = 0 \\
  \text{grd5} : & \; \text{fcharge}(x) = \text{zero} \lor \text{fcharge}(x) = \text{all}
\end{align*}

then

\begin{align*}
  \text{act1} : & \; \text{cell}(x)(b) := \text{cell}(x)(b) - 1 \\
  \text{act2} : & \; \text{cellp}(x)(a) := \text{cellp}(x)(a) + 1 \\
  \text{act3} : & \; \text{fcharge}(x) := \text{zero}
\end{align*}

end

Guards of this event state that the rule is applicable to all unmarked cells with charge \( 0 \), containing at least one object \( b \). If the rule is applied, one object \( b \) is consumed (act1) and one object \( a \) is produced (act2).

Finally, the event corresponding to the communication rule \( [c]^1_2 \rightarrow [c]^0_2 \) is:

**CommunicationRule**
any $x$

where

grd1 : $x \in \text{dom}(\text{cell})$

grd2 : $\text{cell}(x)(c) \geq 1$

grd3 : $\text{charge}(x) = \text{plus}$

grd4 : $\text{mark}(x) = 0$

grd5 : $\text{fcharge}(x) = \text{zero} \lor \text{fcharge}(x) = \text{all}$

then

act1 : $\text{cell}(x)(c) := \text{cell}(x)(c) - 1$

act2 : $\text{charge}(x) := \text{zero}$

act3 : $\text{mark}(x) := 1$

act4 : $\text{skin}(c) := \text{skin}(c) + 1$

act5 : $\text{fcharge}(x) := \text{zero}$

end

Here, $\text{dom}(f)$ represents the domain of the function $f$.

A special event, called update, that is enabled after each step of maximal parallelism, is also needed in order to update the variables before the next computation step. This has the following structure:

Update

when

grd1 : $\forall x \cdot x \in \text{dom}(\text{cell}) \Rightarrow (\text{mark}(x) \neq 0) \lor$

$\neg((\text{cell}(x)(a) \geq 1) \land (\text{charge}(x) = \text{zero})) \land$

$\neg((\text{cell}(x)(b) \geq 1) \land (\text{charge}(x) = \text{zero})) \land$

$\neg((\text{cell}(x)(c) \geq 1) \land (\text{charge}(x) = \text{plus}))$

then

act1 : $\text{cell}, \text{cellp} : \forall x, y \cdot x \in \text{dom}(\text{cell}) \land y \in \text{SYMBOLS}$

$\Rightarrow \text{cell'}(x)(y) = \text{cell}(x)(y) + \text{cellp}(x)(y)$

$\land \text{cellp'}(x)(y) = 0$

act2 : $\text{mark} : \forall x \cdot x \in \text{dom}(\text{cell}) \Rightarrow \text{mark'}(x) = 0$

act3 : $\text{charge}, \text{fcharge} : \forall x \cdot x \in \text{dom}(\text{cell})$

$\Rightarrow \text{charge'}(x) = \text{fcharge}(x) \land \text{fcharge'}(x) = \text{all}$

act4 : $\text{skin}, \text{skinp} : \forall y \cdot y \in \text{SYMBOLS}$

$\Rightarrow \text{skin'}(y) = \text{skin}(y) + \text{skinp}(y) \land \text{skinp'}(y) = 0$

end

An initial model of a P system, in which we are interested only in its evolution, can then be refined by adding details about the state of the computation. At the beginning of every step of maximal parallelism, the system is considered to be in state Running. Another state, Other, is considered as an intermediate state between two such steps. A halting configuration is marked by a transition from state Running to another state, Halt. Finally, in order to keep the number of configurations under control, we can consider upper bounds for the components of a configuration (including the number of membranes) or for the number of times a rule can be applied. When either of these conditions is violated, we consider that the system performs a transition from Running to a fourth state Crash. All these states are implemented using a variable, called state, with four possible values: Running, Other, Halt and Crash. In this
case, all the events in the original model have to be refined - these now become transitions between states Running, and Other. Furthermore, new events have to be introduced for transitions from Running to Halt, Running to Crash, Halt to Halt and Crash to Crash. The context is also extended by adding the set of states. Obviously, the state variable and the extra transitions could have been introduced directly in the original model. However, the use of refinement allows a gradual, more manageable and natural, construction of the model.

4 SubsetSum Model

We consider now a P system with active membranes used to solve the subset sum problem: given a finite set \( A \) with \( n \in \mathbb{N}^* \) elements, where each element has an associated weight, and a constant \( k \), there exists a subset of \( A \) with the sum of the weights of its elements \( k \)?

A P system with active membranes can be constructed to solve this NP-complete problem [16]. We consider the particular case \( n = 2 \) and \( k = 2 \). In this case, the initial configuration of the P system \( \Pi \) contains two membranes and the initial multisets are: \( \{z_0\} \) in the first membrane and \( \{e_0, (ab)^2, x_1, x_2\} \) in the second one.

The set of rules associated with the second membrane contains:

- \( [e_i^0]_2^0 \rightarrow [qp]_2^- [e_i^1]_2^+ \), \( i \in \{0, 1, 2\} \)
- \( [e_i^1]_2^+ \rightarrow [e_i+1_2^0]_2^+ [e_i+1_2^1]_2^+ \), \( i \in \{0, 1\} \)
  (division rules generating a membrane for each subset of \( A \))
- \( [x_0 \rightarrow a0b]_2^0 \)
- \( [x_0 \rightarrow not]_2^+ \)
- \( [x_i \rightarrow x_{i-1}]_2^+ \), \( i \in \{1, 2\} \)
  (evolution rules that calculate in \( a0b \) the weight of a subset)
- \( [qp \rightarrow q0]_2^0 \)
- \( [a0b \rightarrow a0]_2^- \)
- \( [ab \rightarrow a]_2^- \)
  (at the beginning of the checking stage the weight of the subset is represented by the multiplicity of \( a0 \))
- \( [a0]_2^- \rightarrow [ ]_2^2 d \)
- \( [a]_2^0 \rightarrow [ ]_2^- d \)
  (communication rules allowing to compare the occurrences of \( a0 \) and \( a \) in a loop)
- \( [q_{2j} \rightarrow q_{2j+1}]_2^- \), \( j \in \{0, 1, 2\} \)
- \( [q_{2j+1} \rightarrow q_{2j+2}]_2^- \), \( j \in \{0, 1\} \)
  (objects \( (q_i)_{i=1}^{\Gamma_{PP}} \) are used as counters for the loop)
• \([q_5]^2 \rightarrow [ ]^2 Y\)
• \([q_5]^0 \rightarrow [ ]^0 d\)
• \([q_{2j+1}]^2 \rightarrow [ ]^2 d, j \in \{0, 1\}\)
  (communication rules that give an answer to the comparison between \(a_0\) and \(a)\)

The set of rules associated with the skin is:

• \([z_i \rightarrow z_{i+1}]^0, 0 \leq i \leq 9\)
• \([z_{10} \rightarrow d_1 N_0]^0\)
  \((d_1 \text{ and } N_0 \text{ are released into the skin to mark the end of the checking stage})\)
• \([d_1]^1 \rightarrow [ ]^+ d_1\)
• \([N_0 \rightarrow N]^1\)
• \([Y]^+ \rightarrow [ ]^0 Y\)
• \([N]^+ \rightarrow [ ]^0 N\)
  (finally, \(Y\) for “yes” or \(N\) for “no” are sent out in the environment as an answer to the problem)

We consider that the alphabet of the P system can be deduced from the rules.

The model checker ProB can be used to verify properties of the corresponding Event-B model. Table 1 summarizes some of these properties, along with the result produced by the model checker.

<table>
<thead>
<tr>
<th>LTL property</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(G{\text{card}(\text{cell}) &lt; 3})</td>
<td>False</td>
</tr>
<tr>
<td>(G{\text{skin}(N) = 0})</td>
<td>No counterexample found, all nodes have been visited.</td>
</tr>
<tr>
<td>(F{#x:x : \text{dom}(\text{cell}) &amp; \text{cell}(x)(a_0) = 1})</td>
<td>False</td>
</tr>
<tr>
<td>(G{\text{skin}(z_2) &lt; 1})</td>
<td>False</td>
</tr>
<tr>
<td>(G{\exists x:(x : \text{dom}(\text{cell}) &amp; \text{cell}(x)(e_0) &gt; 0 &amp; \text{charge}(x) = \text{zero}) \Rightarrow X{#y,y : \text{dom}(\text{cell}) &amp; \text{cell}(y)(q_p) &gt; 0 &amp; \text{charge}(y) = \text{minus}}}}</td>
<td>No counterexample found, all nodes have been visited.</td>
</tr>
</tbody>
</table>

Table 1: Properties checked for the P system \(\Pi\)

Here “\(\forall\)” “\(\exists\)” and “:” corresponds to the universal quantifier “\(\forall\)” to the existential one “\(\exists\)” and, respectively to the membership operator “\(\subseteq\)”.

In future experiments we will consider bigger values for \(n\) and \(k\) and more complicated properties.
5 Conclusions and future work

Based on rigorous mathematical foundation and allowing high-level modelling, Event-B, Rodin and ProB are strongly supported by the industry.

In this paper we have presented a general framework for modelling P systems with active membranes using Event-B and applied the proposed approach on an example.

Our future work will concentrate on modelling other types of P systems, refinement, simplification, decomposition of models, as well as applying search based techniques for test generation.

References


Abstracts of Poster Presentations
Joining bio-inspired and quantum approaches in computer algebra and computer linguistics

Artiom Alhazov\textsuperscript{1,2}, Liudmila Burtseva\textsuperscript{1}, Svetlana Cojocaru\textsuperscript{1}, Alexander Colesnicov\textsuperscript{1}, Ludmila Malahov\textsuperscript{1}, Yurii Rogozhin\textsuperscript{1}

\textsuperscript{1} Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău MD-2028 Moldova
\{artiom,burtseva,Svetlana Cojocaru,kae,mal,rogozhin\}@math.md

\textsuperscript{2} Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
artiom.alhazov@unimib.it

Abstract. In the paper, the perspectives of computational paradigm, that combines the nature inspired models of computation: bio- and quantum ones, are discussed. The application areas selected to analyze the combined paradigm are computer algebra and computer linguistics that provide very complex problems including NP-complete ones.

1 Introduction

Natural computing is presented as a kind of a bridge between informational and natural sciences. Some domains of natural computing such as evolutionary and neural computing have been investigating during long period. Other domains, such as molecular, membrane, and quantum computing, are much younger. While they are nature-based, molecular and membrane computing use paradigms of biology, whereas quantum computing relies on quantum physics.

We discuss a joint approach integrating membrane and quantum paradigm applied to computer algebra and computer linguistics. These areas are sources of very complex problems, including NP-complete ones.

The research area of membrane computing was motivated by the structure and functioning of a living cell. One of models is based on a cell-like hierarchical arrangement of membranes, while tissue model use non-hierarchical arrangement. Membranes delimit compartments where objects presented as multisets evolve according to the given evolution rules (see [12]).

The quantum algorithms always begin with the preparation of the initial state, then a sequence of quantum operations is applied to the system, and then a measurement fixes the outcome. At the measurement, the resulting superposition of quantum states collapses to a single classical state that is observed. It is important that any quantum operations is reversible.

Quantum computing inevitably needs outside control, namely, at the observation. John H. Reif [13] noted that observation operations in quantum physics
assume implicitly a macroscopic measurement apparatus. A microscopic measurement apparatus will itself be subject to quantum effects. Moreover, if we try to perform the observation operation over a quantum system with \( n \) entangled qubits with quantum tools, we will need even larger unitary quantum system growing in volume exponentially with \( n \). It implies that the quantum computing should be integrated with another paradigm: classical, bio-molecular, etc.

2 Applications in Computer Algebra

The membrane systems are a convenient framework of describing polynomial-time solutions to certain intractable problems in a massively parallel way. Division of membranes makes it possible to create exponential space in linear time, suitable for attacking problems in NP and even in PSPACE. Their solutions by P systems with active membranes have been investigated in a number of papers since 2001, later focusing on solutions by restricted systems.

Concerning mentioned above achievements in NP problems solution, we considered a \#P-complete problem of computing the permanent of a binary matrix. This class of problem allows, in particularly, to generalize the approach from decisional problems to the computational ones. The said problem was successfully resolved by implementation of methods of P systems with active membranes [2].

Besides permanent problem we provided the implementation of basic operations over polynomials used in different algorithms of computer algebra by P system formalism. The P system rules were developed for following operations: to generate a monomial; to calculate the leading coefficient; to replace a monomial; to compare monomials lexicographically; to obtain the residue monomial; to compare non-commutative monomials; for the non-commutative multiplication; to compare strings that represent non-commutative monomials.

We found that several algebraic problems like matrix permanent are suitable for P system formalism, whereas solving of some others by P systems has the same performance as classic computation. From the other hand, some algebraic problems of rings and ideals theory can be represented by quantum computing models [15] with some better performance.

The usage of different computing models (bio- and quantum in our case) in solving of the same problems naturally leads to attempts to develop the hybrid computing models.

In [9], the instance of hybrid computing model is developed by addition of energy to the membranes. The rules can be applied to objects inside membrane if there is enough energy to do it. This makes the model more physically realistic. The model was tested on solving a NP-complete problem (3-SAT). In [14], another instance of hybrid computing model is described. The test task was the one of classical SAT task. Although the algorithm is implemented in the quantum model, bio-molecular operations were used to remove illegal solutions and to reserve legal solutions in the set of all possible solutions that quantum computation resulted.
Let us mention that the attempts to resolve problems related to Gröbner basis, which did not show good performance on P systems, seems to be untested for quantum computing. Therefore these problems are a good testbed for new hybrid models.

We studied a series of problems in Gröbner bases theory (determination if the algebra has the finite dimension, checking if a given set of polynomials forms the Gröbner basis of the given algebra, Gröbner basis calculation).

Let us discuss the Gröbner basis in more details (for proofs, see [6]). The academic bibliography on Gröbner basis can be found in [7].

2.1 Gröbner Basis

Let \( k \) be a commutative field, and \( I \) be an ideal \( I = (f_1, \ldots, f_r) \) in the commutative polynomial ring \( k[x_1, \ldots, x_n] \) over that field. The simplest question is: how to determine if a given polynomial \( f \) is in \( I \)? For one variable \((n = 1)\), we can repeatedly divide \( f \) by each ideal generators, replacing \( f \) with the remainder. If we get zero as the result, then \( f \in I \).

The algorithm for \( n \geq 2 \) also exists but needs some preliminary steps. We need an ordering of all monomials in \( k[x_1, \ldots, x_n] \). (Let us denote monomials as \( x^\alpha = x_1^{\alpha_1} \cdots x_n^{\alpha_n} \).) The ordering is a relation \( > \) on the monomials. The relation should have three properties. At first, it should be a total order: any two monomials can be compared. Secondly, it should be preserved by multiplication of monomials: if \( x^\alpha > x^\beta \), and \( x^\gamma \) is any monomial, then \( x^{\alpha+\gamma} > x^{\beta+\gamma} \). Finally, it should be a well-ordering: every nonempty subset should have a smallest element.

There are many orderings with these properties. The first one is the lexicographic ordering (LEX). Speaking on ordered lists of integers, \( \alpha > \beta \) has its leftmost nonzero entry positive. That is, \( \alpha \) has the bigger entry at the first power in which it disagrees with \( \beta \). Another one is graded lexicographic ordering (DEGLEX), which orders by total degree of monomials, and then lexicographically within each degree.

However, it was proved that the best average runtime for the Buchberger’s algorithm we’ll talk about later is graded reverse lexicographic ordering (DEGREVLEX). It orders monomials by degree. Then, it has \( \alpha > \beta \) iff the rightmost nonzero entry of \( \alpha - \beta \) is negative.

Let us define the multidegree of a polynomial to be the maximal \( \alpha \) in it. The leading monomial is then \( x^\alpha \), the leading coefficient is the coefficient of \( x^\alpha \), and the leading term LT is the lead coefficient times the lead monomial.

Let we have an ideal \( I \) and a set of its generators \( \{f_1, \ldots, f_s\} \) in \( k[x] \). Let us fix a monomial ordering and sort generators by their leading terms. We want to check if a given polynomial \( f \) is in \( I \). The multivariate division algorithm permits us to find a representation \( f = a_1 f_1 + \ldots + a_s f_s + r \). If \( r = 0 \) then \( f \in I \).

The multivariate division repeatedly applies the following until no monomial term of \( f \) is divisible by any of the LT\((f_i)\), or until 0 is gotten as the result:

Take the smallest \( i \) such that the LT\((f_i)\) divides some term of \( f \). Let \( h \) be the largest term of \( f \) with respect to the fixed monomial ordering that is divisible by LT\((f_i)\), and replace \( f \) by \( f - \frac{h}{\text{LT}(f_i)} \cdot f_i \). Each time \( f \) became smaller in the partial
order produced on \( k[x] \) by the fixed monomial ordering. Therefore the process stops.

The result of multivariate division depends both of selected set of generators and of selected ordering. It can be zero in one case and non-zero in the other case.

To avoid this problem, we should not use an arbitrary set of generators. We should extend the starting set of polynomial ideal generators to the Gröbner basis of the ideal \( I \).

Let \( LT(I) \) be the ideal generated by the elements of the form \( LT(f) \) for \( f \in I \). A Gröbner basis (GB) for \( I \) is a set of polynomials \( \{g_1, \ldots, g_m\} \) if the ideal \( (LT(g_1), \ldots, LT(g_m)) \) is equal to \( LT(I) \). Every non-zero ideal has a Gröbner basis. It always exists and is finite in the commutative polynomial rings.

If we perform multivariate division of a polynomial \( f \) with elements of a GB, the resulting remainder will not depend of the basis and of the order we apply the division. This solves completely the problem of ideal membership.

We can have several Gröbner bases for an ideal. If \( G \) is a GB, and there is \( p \in G \) such that \( LT(p) \) is in the ideal generated by the lead terms of the other elements of \( G \), then \( G \setminus \{p\} \) is also a GB. We can use this to reduce any GB. If the reduction is impossible, and the leading coefficients are equal to 1, the GB is called reduced. Each ideal \( I \) has a unique reduced GB independent even of monomials ordering. This solves the problem of ideal equality.

The algorithm of GB calculation was proposed by Bruno Buchberger. For any two polynomials \( p \) and \( q \), let us calculate \( t = \text{lcm}(LT(p), LT(q)) \), where \( \text{lcm} \) stands for the least common multiple. The polynomial \( S(p, q) = \frac{t}{LT(q)}p - \frac{t}{LT(p)}q \) is called S-polynomial. (Here \( S \) stands for syzygy.) S-polynomial is constructed from polynomials \( p \) and \( q \) by annihilating their leading terms.

Having any set of generators \( f = \{f_1, \ldots, f_s\} \) in \( k[x] \), let us start with \( g = f \). For each pair of polynomials \( g_1, g_2 \in g \), let us calculate the remainder \( r \) of multivariate division of \( S(g_1, g_2) \) by \( g \). If the remainder \( r \neq 0 \), expand \( g \) with \( r \) and repeat. The process stops when all pairs \( g_1, g_2 \in g \) produce the remainder 0 at the multivariate division of \( S(g_1, g_2) \) by \( g \). Then \( g \) is a GB.

Faugère proposed a variant of Buchberger’s algorithm where matrix operations are used to perform calculations more quickly.

It is possible to define Gröbner basis for ideals in non-commutative polynomial rings \( k \prec x \). In that case, the GB can be infinite.

Gröbner bases are applicable in a lot of important mathematical and physical problems that can be expressed through systems of polynomial equations. For example, the following very simple criteria are known: the system of polynomial equations is inconsistent iff its GB contains a non-zero constant; the same system has finite number of solutions iff its GB contains polynomials whose leading monomials are powers of only one variable, for each variable.

In mathematics, many problems in differential equations and finite differences, in different domains including but not restricted by graph coloring, integer programming, theorem proving, and cryptography can be solved using GB. In applications, such domains as weather forecast or petroleum production (and
many others) can be exampled. In physics, the non-commutative GB is used as well as commutative in many problems and many domains, for example, in quantum and nuclear physics.

An interesting example is in the paper of Faugère, Hering, and Phan [8], where the GB technique is applied to a biophysical problem of positioning the proteins in living cell membranes.

The GB calculation is exponentially difficult. It involves very large integers as polynomial coefficients, and exponentially grown lists of polynomials. The algorithm is known as difficult to execute in parallel.

Being algorithmically solvable, the GB calculation can be implemented in P system as it covers by the universal machine, but this solution is extremely ineffective.

We tried to implement Buchberger’s algorithm effectively using splicing P systems and P systems with active membranes. Both attempts were unsuccessful because of problem in implementation of the algorithm’s internal recursion. Now we continue to solve this problem using different techniques (for example, we try to implement the algorithm in polymorphic P systems described in [4]). We guess that Buchberger’s algorithm may be effectively implemented in the appropriate hybrid computing model.

3 Applications in Computer Linguistics

We tried to apply P systems with active membranes in computer linguistics. Solving most problems of natural language processing is based on using voluminous linguistic resources. A reasonable approach to obtain efficient solutions is to apply parallelism; this idea has been promoted already in 1970’s. Many of the stages of text processing can be carried out by parallel methods.

We investigated the effectiveness of P systems methods for: dictionary on-line search without indexing and off-line search with indexing; words hyphenation; dictionary updating. We considered also inflections in Romanian language [5], and language generation by non-cooperative membrane systems.

The main results obtained are original methods of representation, creation and management of electronic dictionaries, algorithms for parallelization of the inflection process, and new formalisms for languages generation. We use dictionary represented by a prefix tree and P systems with active membranes that are a convenient framework of describing computations on trees [3]. In [1], we formalized inflection process for the Romanian language using the model of P systems with cooperative string replication rules.

Although the resolving of computer linguistics tasks using P systems is successful, this domain also has some specific features which are hard for the membrane formalism. These issues rely mainly to sequential way of linguistic processing that blocks complete parallelization. On the other hand, the solving of string rewriting task by quantum computational model [10] looks complicated and overloaded of assistant constructions. We also guess that the hybrid model in this case could replace such tricks by P system formalism.
4 Our Suggestions for Hybrid Computing Model

In this section we present our ideas about hybrid model implementation. Generally, the hybrid computer can be demonstrated by the following hypothetic example that uses membrane computations (to prepare data), DNA computations (to process data), and quantum computations (to communicate). The upper macroscopic level uses membrane (P systems) computing model: this is membrane structure with \( n \) levels of depth with \( m \) inner membranes on each level. The structure provides \( m^n \) parallel operations. The deepest level uses the DNA computing model. DNA molecules are providing implementation of rules. On the microscopic level, the interactions between membranes are provided by means of quantum computing model (teleportation or quantum walk depending on particular case).

More practically, we choose the following variants of hybrid model for initial testing.

4.1 Model 1

Model 1 uses input-output of bio-inspired model. Operations are mixed. Problem solution algorithm is based on quantum model. Execution model is mixed.

Mixed execution model means that computation process and execution semantic rules are different for bio and quantum part of algorithm, but control is bio-inspired. For example, an algorithm could be executed as follows: initialize the bio parts of the algorithm to prepare datasets, run the first bio part of the algorithm to prepare quantum state, then initialize the quantum state on the quantum unit, apply unitary transformations, measure the final state, and, finally, evaluate the measurement on the bio computer. If the solution is found, then terminate calculations, else repeat quantum calculations beginning with the initialization of the quantum state.

Inspiration of Model 1 is the fact, that input/output problem is reputed as one of the most significant for quantum computing. The quantum algorithms always begin with the preparation of the initial state, then a sequence of quantum operations is applied to the system, and then a measurement fixes the outcome. At the measurement, the resulting superposition of states collapses to a single classical state that is observed.

Another reason for trying Model 1 was presented in [14, Sec. 2]. In this work bio-inspired methods used to remove illegal solutions from plural quantum computation results. The perspectives of Model 1 are shown in this work by theoretical investigations about implementation of bio model algorithms by quantum model. Practical experiments using nuclear magnetic resonance technique as quantum computing realization show the possibility of physical testing of hybrid model.

4.2 Model 2

Model 2 also uses input-output of bio-inspired model and mixed operations. But in this case problem solution algorithm and execution model are bio-inspired.
The inspiration of this model was the fact that events inside the living cell have both quantum and bio nature. So, the mixed operation set will include execution rules both of bio-inspired model and of quantum one.

The example of Model 2 can be taken from our membrane computing solution of matrix permanent problem [2, Sec. 2]. The membrane rules set applied in permanent problem solution includes the change of membrane polarization. This operation can be replaced by quantum transfer of membrane states.

Linguistic task of dictionary search mentioned in Sec. 3 can be implemented using Model 2. This algorithm includes the search of membrane marked by the due letter. We can apply the quantum search that is extensively used to solve combinatorial problems for this step.

The test of Model 2 is also induced by the work [11] where the usage of photon-based events of biologic molecules as physical implementation of quantum computers is tested. This research shows the possibility of development of real wetware for hybrid models.

5 Conclusions

The domains of computer algebra and computer linguistics are characterized by problems of the high computational complexity. Therefore, the interest in effective methods of their solution is justified. The results of our research demonstrate that natural computing make an effective mechanism to solve problems in these domains, in particular, for non-commutative algebras where the calculations can be infinite.

In addition to high computational complexity of both mentioned domains there are the set of common operations because words can be interpreted as non-commutative monomials.

Summarizing our results of application of P systems methods in domains of computer algebra and computer linguistics and the known results in the same domains using the quantum approach, we can ascertain directions of future research.

Both successful and unsuccessful attempts are useful, giving the picture of advantages and disadvantages of particular computing model in application to these domains. The exploration of computing model structure, that is obtained from comparative analysis mentioned above, allows to build new models employing the features of different natural computing models for successful solving of hard computational tasks.

References


Creating a P system from a Trace using Genetic Algorithms

Alex Ciobanu, Florentin Ipate

Department of Computer Science, University of Pitesti
Str. Targu din Vale 1, 110040 Pitesti, Romania
florentin.ipate@ifsoft.ro

Abstract. This paper continues the work of developing a P system using genetic algorithms, by introducing a fitness function able to resolve more complex problems than previous attempts.

Keywords: P systems generation, genetic algorithms, evolutionary approach.

1 Introduction

The field of membrane computing, which deals with distributed and parallel computing models called P systems, has been a rapidly growing research area in the last ten years. Initially coined by Gheorghe Păun in [1], P systems have been intensively studied from a computational perspective as many variants have been introduced and investigated and a substantial set of applications have been identified and modeled with such systems [3]. Unfortunately the process of creating P systems fit for particular tasks is a non trivial one. We endeavor to discover optimal P system using genetic algorithms from the trace of a P system. We define the trace of a P system to be a sequence of multisets detailing the objects in the membrane(s) at sequential evolutions. For the purposes of this article we will only be considering the basic type of P systems with evolution / communication rules, the cell-like P systems. This article tries to extend on the work done in article [2] by developing a different method for genetically generating P systems.

2 The Problem

In our quest to auto generate a P system using a specification of its behavior we will define a P system as: \( \Pi = (\mathcal{V}, W_1, r_1, \ldots, r_n) \) where \( \mathcal{V} \) is the alphabet (finite and nonempty set of objects) ; \( W_1 \) is a multiset over \( \mathcal{V} \), describing the multisets of objects initially placed in the membrane; \( r_1, \ldots, r_n \) is a finite set of evolution / communication rules. An evolution rule is of the form \( r : v \rightarrow v' \) where \( v \) is a single item object from \( \mathcal{V} \) and \( v' \) is a multiset of objects over \( \mathcal{V} \); in our examples, \( v' \) will contain 1-3 objects.
We use single membrane P system as any multi membrane simple P system can always be flattened to a single membrane P system as per article [5]. Our target is to take a trace of a P system, for example $s \Rightarrow ab \Rightarrow bc \Rightarrow cc$ and calculate a P system for which one possible evolution of the P system would generate the objects outlined (or a super set of those objects) at each step of the evolution.

We define the optimal solution to be a P system where:

1. all of the configurations of the trace are satisfied.
2. the number of rules is minimal, e.g. a P system with 4 rules is preferred to one with 5 rules
3. rules which adhere strictly to the requirements of the trace are preferred to rules which produce superfluous objects

Precedence, in case of conflicts between the requirements, is given in the order enumerated above.

3 The Setup

Genetic algorithms all function on the same basic principle of chromosomes which are composed of genes, cross-combined and killed based on a fitness function [6]. To produce a P system using these principles we defined a gene to represent a P system rule. The data structure is composed of two multisets leftObjects and rightObjects where the rule produced would be $leftObject \rightarrow rightObjects$

Each chromosome in the population represents a P system (a set of rules). For our experiments we used fixed length chromosomes where the number or genes (rules) was established a priori from a best guess to the number of rules required to satisfy the trace. As the number of rules in our P system is fixed we give special notice to rules in the P system that are superfluous as they could be eliminated without effecting functionality, as such helping achieve point two from the section above.

In genetic algorithms the fitness function is used to determine which chromosomes are kept, crossed over, and which are eliminated from the population. In our case a higher fitness value represents a chromosome which is closer to an optimal solution. For each step of the trace the rules applicable in that configuration of the P system must produce all of the objects outlined in the trace. If they are all produced the fitness value increases by 1 unit, but if only a subset of the required objects are produced the fitness value increases by the percentage of objects produced e.g. if 3 out of 5 objects are produced the fitness value increases by 0.6 (60%). Even if the rule does not produce any useful objects, a small increase in fitness is given if the rule can be applied in the membrane. This increases the probability of applicable rules creation in future generations. For a P system that satisfies the entire trace an increase in fitness of 1 unit is given for each rule in the P system that is not used in order to favor the P system with fewer rules. The full algorithm is (inspired from [4]):

for each step in trace do
while there exist rules useful to apply and applicable do
maximally apply these rules with precedence to rules with highest number of useful objects
end while
if the applied rules produce all objects needed for the step in trace then
increase fitness value by 1
if any of the applied rules produce superfluous objects then
decrease fitness value by a small amount
end if
else
if some of the needed objects were created then
increase fitness value by the percent of needed objects created
else
increase fitness value by small amount for each rule that can be applied
end if
return fitness value
end if
end for
for each rule in P system not used in fulfilling trace do
increase fitness value by 1.
end for
return fitness value

This fitness function guides the algorithm to the discovery of an optimal P system as per our definition. To give an example, if our trace was
\[ s \Rightarrow ab \Rightarrow bc \Rightarrow cc \]
and the P system given to test was
r1: \{s \rightarrow ab\}
r2: \{b \rightarrow c\}
r3: \{c \rightarrow a\}
the first rule would be flagged as applicable in the first step of the trace \(s \Rightarrow ab\) and the rule would be applied once satisfying the requirements for the first step in the P system with no penalty for superfluous objects. Fitness value after the first step would be 1. For the second step of the trace \(ab \Rightarrow bc\) only the second rule would be applicable and applicable once. This rule application would produce 50% of the object required in that step hence a fitness value of 0.5 is added to the running total. Since the second step was not fully realized the algorithm exits with a fitness value of 1.5 for this particular P system.

The most important part of this algorithm, in our case, is the mutation function. The search space of genes (P system rules) is large and can not be covered by initial generation of random genes (at the initialization of the population) in the genetic algorithm. The mutation function is the only way to introduce new genes (P-system rules) into the population to help find the ideal chromosome (P system) for a particular trace. In our application a special lock was added to
every gene (rule) in a chromosome (P system) which the fitness function deemed applicable at any step of the trace. All other genes where free for random mutation. In other words any rule that could help achieve the trace was kept in the population, all other genes were thrown away or better said mutated. Mutations randomly generate a new gene (P-system rule). The mutation checks if the leftObject was flagged as useful, if so it keeps it, otherwise it randomly selects another member from the alphabet for leftObject. Next, the mutation generates 1-3 more objects to be placed into rightObject. Upon completion the mutation has generated a brand new rule for our P system. The crossover function was used as defined in JGAP [6] and no attempts were made to modify or improve its functionality.

4 Experimental Results

For the first experiment performed, 10 consecutive attempts were made to generate the P system based on the trace $s \Rightarrow ab \Rightarrow bc \Rightarrow cc$. The average number of evolutions required to achieve optimal results were recorded. Do note, the heuristic nature of the algorithm introduces a significant amount of instability in results hence consecutive executions may produce slightly different results. In the experiment the effect of mutation rate was verified. The optimal solution discovered (as per our definition of optimal from section 2) is a two rule P system:

\[
\begin{align*}
   r1: & \{ s \rightarrow ab \} \\
   r2: & \{ b \rightarrow bcc \}
\end{align*}
\]

To discover this solution we ran the experiment several times for 1000 generations and recorded the best solution we could achieve and its corresponding fitness value. We recorded the average number of generations required to achieve the same fitness value, and the success rate of achieving this result. A maximum of 1000 evolutions was allowed

<table>
<thead>
<tr>
<th>Mutation rate</th>
<th>100%</th>
<th>60%</th>
<th>25%</th>
<th>10%</th>
<th>1%</th>
<th>0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success rate</td>
<td>100%</td>
<td>90%</td>
<td>80%</td>
<td>20%</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td>Number of Evolutions</td>
<td>86</td>
<td>146</td>
<td>328</td>
<td>804</td>
<td>825</td>
<td>1000</td>
</tr>
</tbody>
</table>

We will note here, the number of evolutions also includes non successful attempts at finding a solution. In most cases the number of evolutions required to produce an optimal solution was either a very low number (10 to 100 evolutions) or no result was found. This can be easily explained by the randomness factor. Either good genes were created by luck at the beginning or no good genes would ever be created if the mutation rate was low.

We also performed an experiment on the trace $ag \Rightarrow aadg \Rightarrow abbig \Rightarrow bcchhi \Rightarrow cddhhh$ to discover the algorithm’s performance under more complex conditions. Here we allowed for 10000 evolutions and 8 rules in the P system. Out of 5 experiments, 3 were successful in creating a P system which satisfies the trace, with one of the best solutions being:
5 Conclusions

The algorithm developed shows how P systems can be generated using genetic algorithms based on a trace of the P system. However, useful this heuristic algorithm has a significant component of randomness where results can be achieved in 5 evolutions or 500 evolutions with no modification to the algorithms or parameters. Through experimentation, optimal P systems were either discovered relatively quickly or not at all. This is due to the heavy reliance on the mutation algorithm in creating the rules necessary for crafting the optimal P system. We observed that as the generations progressed, more and more rules were locked into a non-mutation state hence the probability for good rules to be created is reduced as fewer rules are being mutated. For future work more stability in the generation of P systems is to be sought.

Acknowledgments. This work was supported by CNCSIS - UEFISCSU, project number PNII - IDEI 643/2008.

References

A computational model to study the dynamics of Pyrenean Newt (*Calotriton asper*)

Maria Angels Colomer\(^1\), Albert Montori\(^2\), Isart Gaspa\(^3\), and Cristian Fondevilla\(^4\)

\(^1\) Dpt. of Mathematics, University of Lleida
Av. Alcalde Rovira Roure, 191. 25198 Lleida, Spain
{colomer,cfondevilla}@matematica.udl.cat

\(^2\) Dpt. de Biologia Animal (Vertebrats), University of Barcelona
Av. Diagonal, 645. 08028 Barcelona, Spain
amontori@ub.edu

Abstract. Using probabilistic functional multi-environment P-System we model the dynamics of a newt population living in two streams. The newt species is threatened by anthropic factors. The availability of a validated model to study the population’s dynamics will enable managers to take decisions in order to conserve the species.

Keywords: Newt, trout, dynamic, probabilistic, multi-environment

1 Introduction

The Pyrenean Newt (*Calotriton asper*) is a species listed as of especial interest in Spain. The endemic species, located only in the Pyrenees, is abundant in its natural habitat. Anyhow the population is small and subjected to elements that can be harmful for its normal development, such as the introduction of species for the use in fishing sports and the exploitation of torrents [13].

The species is included in the category Near Threatened (NT) according to the IUCN because it occupies a surface which hardly exceeds 20 000 km\(^2\) and its habitat is in decline. According to these parameters, the species could almost be categorized as vulnerable. In Spain it is protected by Royal Decree 1980 and 1986.

In the Pyrenees newts are especially threatened by the introduction of trout for fishing. The habitat is not appropriate for the development of trout, so its reproduction is difficult. Nevertheless the presence of trout causes a significant decrease in the reproductive success of newt, as the trout feed newts’ eggs and larvae. Besides the risk of sporadic floods that occur in the area can be a difficult situation for the newts.

The world-wide decline of amphibian populations is a fact widely accepted. Climatic changes, habitat loss, ultraviolet-B (UV-B) radiation, acid rain and emerging diseases are considered to be reasons for this descent [1, 4, 15].
Then again, Biek [2] considers that current research, inventory and monitoring efforts should be supplemented with demographic studies, so that quantitative analyses can be applied to a wider range of species and life-history groups. Organisms inhabiting running waters are frequently exposed to fluctuations and strong changes of hydrological conditions [16]. Species and communities living in these habitats recover usually quickly from catastrophic events of strong hydraulic discharge, since many species seek shelter in benthic refugia, such as stones, debris dams, woody debris or fissures in the rocky river bed [3, 8, 9]. Nevertheless, drift has been described as one of the most important factors modifying the population’s structure of running water communities [8, 9].

Disposing of a computer model to simulate the dynamics of species under different scenarios can help to improve the management of streams and their environment, besides protecting the species.

Mathematical models usually use differential equations. Such a model does not allow the study of complex interrelated problems. P-systems make this possible, as they are bio-inspired models, based on the functioning of cells. In the field of P-systems we have already modelled three real problems: Scavenger birds, zebra mussels and Pyrenean Chamois. In all three cases the results have been satisfactory [7], [5]. P-systems are considered a good tool in order to study population dynamics.

In this work we will propose a model for the Pyrenean Newt which will take into account the basic processes of reproduction, mortality, movement of the newt in the streams and flood risk. The modelling of natural processes is usually hard, given the large number of variables and processes that are involved. Besides, these processes are inter-related.

In the following section a description of the area of application and the species’ basic biology will be given. After defining a model we present a simulator developed with MeCoSim. Finally we will compare the results of the model with real data.

2 Species’ Biology

The work focuses on the newt population in the Serra del Cadí, situated in the Pre-Pyrenees (NE Spain). In the study area there are 38 main streams, see fig.1. In this work we will only model the dynamics of species living in two streams: Pi and Ingla.

The newt is an endemic species, which prefers streams with steep slopes and no aquatic vegetation at the margins or at the bottom of the river. The newt avoid loamy soils, like stones and gravel soils [6]. They live in streams with a water temperature exceeding 16 °C [3].

Pyrenean newt live in some of the smaller streams that flow into the river Segre. The studied population is a typical example of the species’ habitat which is affected by strong population fluctuations caused in most cases by severe floods.

Montori ([10]) describes three habitat types for this species: medium and high mountain streams, high mountain lakes and underground streams, whereat
the first is the typical habitat of the species. In this environment larvae are able to pass up to two winters into the water, though the usual case is the metamorphosis and the terrestrial life after one winter. Having reached sexual maturity at the age of three or four, the amphibians return to the water and live in both terrestrial and aquatic environments. Individuals older than 9-11 years remain in turn in the water only, until they exceed 20 years.

The amount of laid eggs per clutch differs in the consulted references. Clergue-Gazeau [6] specifies it as about 15-25 eggs per clutch, while Montori [12] counts 39.87 eggs [13]. There is a positive correlation between female’s size and the number of eggs in the clutch. Montori [12] describes the following relationship:

$$y = 2.11 \cdot x - 95.28$$  \hspace{1cm} (1)

Whereat $y$ is the body size and $x$ is the number of eggs per clutch.

Montori, [11], defines the function of growth with the following expression:

Female:

$$y = 73.66 \cdot \left(1 - e^{-0.145(x+2.79)}\right)$$  \hspace{1cm} (2)

Male:

$$y = 74.68 \cdot \left(1 - e^{-0.179(x+1.79)}\right)$$  \hspace{1cm} (3)

Whereat $y$ is the body size and $x$ is the age.

Using the values obtained by applying equations (1) and (2) and logarithmic regression we obtained the following expression for the number of eggs per clutch.
and the age of newt.

\[ y = 6.8800 \cdot \ln(x) + 58.167 \quad (R^2 = 0.982) \]

Whereat \( y \) is the number of eggs per clutch and \( x \) is the age.

The oldest newt specimen found in the studied area were 26 years old [10]. In other studies the located animals were up to 29 years old [13]. The sex ratio is 1:1 [10].

The nutrition of the Pyrenean newt consists basically of aquatic organisms, in particular fish larvae [10]. Food is not a limiting factor for the newt population in the study area.

According to estimations based on the model of Peterson and Jolly-Seber, the population contains between 3600 and 5596 adult specimen in 1.5 km of river [10]. Data records display the disappearanace of populations in lakes after the introduction of trout (Salvelinus fontinalis). In streams the populations have decreased due to the introduction of river trout (Salmo trutta fario). Any medium-sized aquatic species can be a potential predator for the newt [13]. The population is furthermore affected by floods, though a recovery is possible due to the terrestrial life of sub-adults, which act as a reservoir population [13].

The newt are primarily sedentary but may perform displacements of about 1000 meters while they live in water and soil. Using the lognormal function with mean 400 m we estimate the probability of movement between streams.

Intraspecific territoriality was not observed, as several individuals share one shelter [13].

There are several factors that affect the life of the newt at the same time: stream temperature, precipitation, introduced fish species, the reproduction capacity of females, etc. Each of these parameters can be studied separately, but we need a model that can work with all of them in parallel, since all these parameters are interrelated. It is important to dispose of a model that allows us to predict with certainty the population dynamics, especially of endangered species.

3 The model

In order to model the ecosystem we use a multi-environment functional probabilistic \( P \) system with active membranes [7] of degree (4, 2) (four membranes and five environments), taking \( T \) time units (simulation years). We model the population dynamics of 2 animal species and 2 environments.

\[ (G, \Gamma, \Sigma, R_E, \Pi, \{ f_{r,j} : r \in R_H, 1 \leq j \leq 2 \}, \{ M_{i,j} : 0 \leq i \leq 3, 1 \leq j \leq 2 \}) \]

Where:

1. \( G = (V, S) \) is a direct graph such that \((x, x) \in V \) for each \( x \in V \). Let \( V = \{ e_1, e_2 \} \) whose elements are the environment.

2. The membrane structure is

\[ \mu = [[1] \quad [2] \quad [3] \quad 0] \]

The initial configuration is
Environment : \{F, RI_1, RI_2\}
Membranes : \begin{align*}
M_0 &= \{R_0, X_{ij} : 1 \leq i \leq 2, 1 \leq j \leq g_5\}, \\
M_i &= \{\emptyset\}, 1 \leq i \leq 3
\end{align*}

3. The working alphabet of the P system is
\[ \Gamma = \{X_{ij}, Y_{ij}, Z_{i,j}, W_{i,j}, WM_{i,j} : 1 \leq i \leq 2, 0 \leq j \leq g_5\} \cup \{L_i, LM_i, 1 \leq i \leq 2\} \cup \{R_i, 0 \leq i \leq 10\} \]

Objects \( X_{ij}, Y_{ij}, Z_{i,j}, W_{i,j}, WM_{i,j} \) represent the same animal but in different states. Objects \( L_i, LM_i \) represent larvae and \( R_i \) is a counter.

4. The environment alphabet is
\[ \Sigma = \{WM'_{i,j,v} : 1 \leq i \leq 2, 0 \leq j \leq g_5, 1 \leq v \leq 2\} \cup \{LM'_{i,v} : 1 \leq i \leq 2, 1 \leq v \leq 2\} \cup \{V M_{i,v} : 1 \leq i \leq 2, 1 \leq v \leq 2\} \cup \{LM_i : 1 \leq i \leq 2\} \]

Objects \( WM' \) and \( LM' \) are associated to the animals.

5. The model is structured by modules, see Fig.2, the set of rules \( R_E \) and \( R_H \) that applied in each module are:

In order to synchronise the model we use the object \( R_i \)
\begin{align*}
& r_1 \equiv [R_i \rightarrow R_{i+1}]^0, 0 \leq i \leq 13, i << 4, i << 7, i << 9, i << 10, i << 11 \\
& r_2 \equiv [R_i]^{13} \rightarrow [R_0]^1 \\
& r_3 \equiv [R_7S_1]^{10} \rightarrow [R_8]^{2} \\
& r_4 \equiv [R_7S_2]^{10} \rightarrow [R_8]^{3} \\
& r_5 \equiv [R_7S_3]^{10} \rightarrow [R_8]^{4} \\
& r_6 \equiv [R_7N] \rightarrow [R_9]^{10} \\
& r_7 \equiv [R_9]^{10} \rightarrow [R_{10}]^{13} \\
& r_8 \equiv [R_{10}] \rightarrow [R_{11}]^{13} \\
& r_9 \equiv [R_{11}]^{13} \rightarrow [R_{12}]^{10} \\
& r_{10} \equiv [R_{14}]^{10} \rightarrow F, RI_1, RI_2[R_0]^{10}
\end{align*}

Simulation flood module

In the study area exists the possibility of floods due to high rainfall, the model simulates the existence of this climatic phenomenon using the object \( F \). There is a significant correlation between the environments of the variable presence or absence of floods in the study area. The object \( F \) located in the environment \( e_1 \) can evolve into the objects \( S_i, 1 \leq i \leq 3 \) or \( N \) while \( F \) objects, located in the environment \( e_2 \), is dissolved. Information obtained from the objects \( F \), which have evolved within the environment \( e_1 \), are sent to other environments.

If flooding occurs between March and September, the effect on the population is more important than if it takes the rest of the year. In the case that take account in the first period the object \( F \) evolve to object \( S_1 \), if given in the second to \( S_2 \) and if it occurs in both periods to \( S_3 \).
The specimens that survived the predation come to the skin membrane

- Reproduction module
  - Males that do not reproduce:
    \[ r_{14} \equiv [X_{i,j} \overset{(g_{6,i},1)}{\longrightarrow} Y_{i,j}]^{0}_{i,j} \]
  - Females of reproductive age
    * That reproduce
    \[ r_{15} \equiv [X_{i,j} \overset{(g_{6,i},1)}{\longrightarrow} Y_{i,j}]^{0}_{i,j} \]
    * That do not reproduce
    \[ r_{16} \equiv [X_{i,j} \overset{(g_{6,i},1)}{\longrightarrow} Y_{i,j}]^{0}_{i,j} \]
  - Adult who have stopped reproduce
    \[ r_{17} \equiv [X_{i,j} \overset{(g_{6,i},1)}{\longrightarrow} Y_{i,j}]^{0}_{i,j} \]
  - Young animals
    \[ r_{18} \equiv [X_{i,j} \overset{(g_{6,i},1)}{\longrightarrow} Y_{i,j}]^{0}_{i,j} \]

- Feeding module
  Food is not a limiting factor. We will only take into account the consumption of larvae and juveniles of newt by trout.

When there are trout, object \( a \) is generated and the polarization of membrane 1 changes to negative

\[ r_{19} \equiv Y_{2,j}^{1} \overset{0}{\longrightarrow} Z_{2,j}[a]^{1} \]

Trout predates both larvae and juveniles of newt.

\[ r_{20} \equiv L_{1}^{1} \overset{0}{\longrightarrow} [L_{1}]^{0} \]

\[ r_{21} \equiv Y_{1,j}^{1} \overset{0}{\longrightarrow} [Y_{1,j}]^{0} \]

\[ r_{22} \equiv [a]^{1} \overset{0}{\longrightarrow} [a]^{0} \]

The specimens that survived the predation come to the skin membrane.
A computational model to study the dynamics of Pyrenean Newt

**Module natural mortality**

Can be found in the same step objects associated with animals in the form of \( Z_{i,j} \) and \( Y_{i,j} \)

- **Larval phase**

  \[
  r_{28} \equiv L_i[1]_3^+ \xrightarrow{m_{i,1}} [0]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

  \[
  r_{29} \equiv L_i[1]_3^+ \xrightarrow{1-m_{i,1}} [L_i]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

- **Young animals that live in the soil and water**

  \[
  r_{30} \equiv Z_{i,j}[1]_{3,1}^+ \xrightarrow{m_{i,j}} [0]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{31} \equiv Z_{i,j}[1]_{3,1}^+ \xrightarrow{1-m_{i,j}} [W_{i,j}]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

- **First step adult animals**

  \[
  r_{32} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{m_{i,j}} [0]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

- **Second step adult animals**

  \[
  r_{33} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{1-m_{i,j}} [W_{i,j}]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

- **Animals that have reached life expectancy**

  \[
  r_{34} \equiv Z_{i,j}[1]_{3,1}^+ \rightarrow [0]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

  \[
  r_{35} \equiv Y_{i,j}[1]_{3,1}^+ \rightarrow [0]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

- **Module mortality due to floods**

  \[
  r_{36} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{m_{i,j}} [0]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{37} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{1-m_{i,j}} [W_{i,j}]_{3,1}^0, \quad g_{2,1} \leq j < g_{3,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{38} \equiv Z_{i,j}[1]_{3,1}^+ \xrightarrow{m_{i,j}} [0]_{3,1}^0, \quad g_{2,1} \leq j < g_{4,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{39} \equiv Z_{i,j}[1]_{3,1}^+ \xrightarrow{1-m_{i,j}} [W_{i,j}]_{3,1}^0, \quad g_{2,1} \leq j < g_{4,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{40} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{m_{i,j}} [0]_{3,1}^0, \quad g_{2,1} \leq j < g_{4,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{41} \equiv Y_{i,j}[1]_{3,1}^+ \xrightarrow{1-m_{i,j}} [W_{i,j}]_{3,1}^0, \quad g_{2,1} \leq j < g_{4,1}, 1 \leq i \leq 2 
  \]

  \[
  r_{42} \equiv Z_{i,j}[1]_{3,1}^+ \rightarrow [0]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

  \[
  r_{43} \equiv Y_{i,j}[1]_{3,1}^+ \rightarrow [0]_{3,1}^0, \quad 1 \leq i \leq 2 
  \]

  \[
  r_{44} \equiv [L_i]_{3}^0 \rightarrow [L_i]_{3}^0, \quad 1 \leq i \leq 2 
  \]

  \[
  r_{45} \equiv [W_i]_{3}^0 \rightarrow [W_i]_{3}^0, \quad 1 \leq j < g_{4,1}, 1 \leq i \leq 2 
  \]
• If flood is only one of the two periods.

\[ r_{46} \equiv L_i \big[ \big]_m \rightarrow \big[ L_i \big]_m^0, \quad 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{47} \equiv W_{i,j} \big[ \big]_m \rightarrow \big[ W_{i,j} \big]_m^0, \quad g_{3,i} \leq j \leq g_{6,i}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{48} \equiv \big[ L_i \big]_m^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ L_i \big]_m^0, \quad 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{49} \equiv \big[ L_i \big]_m^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ L_i \big]_m^0, \quad 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{50} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad g_{2,i} \leq j \leq g_{3,i}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{51} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad g_{2,i} \leq j \leq g_{3,i}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{52} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad g_{3,i} \leq j \leq \frac{g_{4,i} - 1}{2}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{53} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad g_{3,i} \leq j \leq \frac{g_{4,i} - 1}{2}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{54} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad \frac{g_{4,i} - 1}{2} \leq j \leq g_{4,i}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

\[ r_{55} \equiv \big[ W_{i,j} \big]_m^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_m^0, \quad \frac{g_{4,i} - 1}{2} \leq j \leq g_{4,i}, 1 \leq i \leq 2, 2 \leq m \leq 3 \]

• Flooding occurs in the two periods.

\[ r_{56} \equiv L_i \big[ 1 \big]_2^+ \rightarrow \big[ L_i \big]_2^0, \quad 1 \leq i \leq 2 \]

\[ r_{57} \equiv W_{i,j} \big[ 1 \big]_2^+ \rightarrow \big[ W_{i,j} \big]_2^0, \quad g_{3,i} \leq j \leq g_{6,i}, 1 \leq i \leq 2 \]

\[ r_{58} \equiv \big[ L_i \big]_2^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ L_i \big]_2^0, \quad 1 \leq i \leq 2 \]

\[ r_{59} \equiv \big[ L_i \big]_2^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ L_i \big]_2^0, \quad 1 \leq i \leq 2 \]

\[ r_{60} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad g_{2,i} \leq j \leq g_{3,i}, 1 \leq i \leq 2 \]

\[ r_{61} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad g_{2,i} \leq j \leq g_{3,i}, 1 \leq i \leq 2 \]

\[ r_{62} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad g_{3,i} \leq j \leq \frac{g_{4,i} - 1}{2}, 1 \leq i \leq 2 \]

\[ r_{63} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad g_{3,i} \leq j \leq \frac{g_{4,i} - 1}{2}, 1 \leq i \leq 2 \]

\[ r_{64} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad \frac{g_{4,i} - 1}{2} \leq j \leq g_{4,i}, 1 \leq i \leq 2 \]

\[ r_{65} \equiv \big[ W_{i,j} \big]_2^0 \xrightarrow{1-m_{f_{i,m}^{-1}}} \big[ W_{i,j} \big]_2^0, \quad \frac{g_{4,i} - 1}{2} \leq j \leq g_{4,i}, 1 \leq i \leq 2 \]

Module of movement by environment

\[ r_{66} \equiv W_{i,j} \big[ 1 \big]_4^+ \rightarrow \big[ W_{i,j} \big]_4^0, \quad 1 \leq j \leq g_{4,i}, 1 \leq i \leq 2 \]

\[ r_{67} \equiv L_i \big[ 1 \big]_4^+ \rightarrow \big[ L_i \big]_4^0, \quad 1 \leq i \leq 2 \]

\[ r_{68} \equiv W_{i,j} \big[ 1 \big]_4^+ \rightarrow WM_i \big[ \big]_4^0, \quad 1 \leq j \leq g_{4,i}, 1 \leq i \leq 2, \]

\[ r_{69} \equiv \big[ L_i \big]_4^0 \rightarrow LM_i \big[ \big]_4^0, \quad 1 \leq i \leq 2 \]

\[ r_{70} \equiv \big[ WM_i,g_{3,i} \big]_k^0 \xrightarrow{m_{f_{i,m}^{-1}}} \big( WM'_i,g_{3,i} \big) \big[ \big]_k^0, \quad 1 \leq i \leq 2, 1 \leq k \leq 2, 1 \leq v \leq 2 \]

\[ r_{71} \equiv \big( WM'_i,g_{3,i} \big) \big[ \big]_k^0 \rightarrow \big( WM'_i,g_{3,i} \big) \big[ \big]_k^0, \quad 1 \leq i \leq 2, 1 \leq j \leq g_{4,i}, j < g_{3,i}, 1 \leq k \leq 2 \]

\[ r_{72} \equiv \big[ LM_i \big]_k^0 \rightarrow LM_i' \big[ \big]_k^0, \quad 1 \leq i \leq 2 \]
The definition of the parameters used in the model are displayed in table 1:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_{1,i}$</td>
<td>Percentage of females</td>
</tr>
<tr>
<td>$g_{2,i}$</td>
<td>Spend time to metamorphosis</td>
</tr>
<tr>
<td>$g_{3,i}$</td>
<td>Age when the terrestrial and aquatic phase finalize</td>
</tr>
<tr>
<td>$g_{4,i}$</td>
<td>Life expectancy</td>
</tr>
<tr>
<td>$g_{5,i}$</td>
<td>Age of female reproduction start</td>
</tr>
<tr>
<td>$g_{6,i}$</td>
<td>Age of female reproduction ending</td>
</tr>
<tr>
<td>$g_{7,i}$</td>
<td>Percentage the females that reproduce</td>
</tr>
<tr>
<td>$m_{1,i}$</td>
<td>Probability of natural death in larvae phase</td>
</tr>
<tr>
<td>$m_{2,i}$</td>
<td>Probability of natural death in terrestrial and aquatic phase</td>
</tr>
<tr>
<td>$m_{3,i}$</td>
<td>Probability of natural death of first phase of adults</td>
</tr>
<tr>
<td>$m_{4,i}$</td>
<td>Probability of natural death of second phase of adults</td>
</tr>
<tr>
<td>$m_{5,i}$</td>
<td>Probability of death due to floods in larval phase</td>
</tr>
<tr>
<td>$m_{6,i}$</td>
<td>Probability of death due to floods in terrestrial and aquatic phase</td>
</tr>
<tr>
<td>$m_{7,i}$</td>
<td>Probability of death due to floods in first adult phase</td>
</tr>
<tr>
<td>$m_{8,i}$</td>
<td>Probability of death due to floods in second adult step</td>
</tr>
<tr>
<td>$m_{9,i}$</td>
<td>Number of animals species $i$ introduce in the environment $k$ per year</td>
</tr>
<tr>
<td>$m_{10,i}$</td>
<td>Probability of flood March - September</td>
</tr>
<tr>
<td>$m_{11,i}$</td>
<td>Probability the newt move between the environment $k$ to $v$ (terrestrial-aquatic phase)</td>
</tr>
</tbody>
</table>

The model is formed by six modules, its scheme is presented in (see fig. 2). In order to simulate one year it’s necessary to run fifteen steps.

4 Simulator and Results

We define the simulator using MeCoSim [14]. The simulator incorporates two different types of menus: the fixed simulator menus, with which it is possible to carry out the simulation; and the specific menus, created to introduce the initial parameters of the ecosystem. The input menu can be divided into several sub-menus depending on the needs.

The first option are the parameters: the biologic, reproduction and mortality parameters. The second option is dedicated to the population, the third to
possible movements between environments and the last to the probability of a flood.

To validate the model we have information of the populations in the stream of Pi for the years 1982 to 2002. Populations correspond to the animals observed in aquatic environments.

In fig. 3 we show the results obtained. In the Pi stream we see that the model has a similar behavior in the number of animals and irregularities that show the actual data. It should be noted that the existence of floods is a randomly simulated data by the model so it is unlikely to match the year that has happened in reality. The streams model is part of a group of 35 streams of the Serra del Cadí, so there are movements of animals between the streams studied and the rest. When we introduce in the model the rest of streams we expect an improvement in the results.

5 Conclusions

We present a model for the study of an endemic species that is threatened primarily by the introduction of trout for sport fishing. With this and previous works it is shown that a wide variety of natural processes can be successfully modelled by means of bio-inspired models. Furthermore P system based models offer a modularity that makes them very useful, as it makes it possible to improve and extend the model presented in this paper easily and quickly.
The model results presented are very encouraging and are expected to improve with the introduction of streams and the additional factors, basically human, which affect the dynamics of the species studied.
References

Plant communities dynamics model using P systems

Cristian Fondevilla¹, Maria Angels Colomer¹, Federico Fillat², and Luis Valencia-Cabrera³

¹ Dpt. of Mathematics, University of Lleida
Av. Alcalde Rovira Roure, 191. 25198 Lleida, Spain
{colomer,cfondevilla}@matematica.udl.cat

² Conservation of Biodiversity in Mountain Ecosystems
IPE-CSIC
Avda. Rgto. Galicia, s/n. 22700 Jaca (Huesca), Spain
ffillat@ipe.csic.es

³ Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
lvalencia@us.es

Abstract. In this work we present a P system based model of the ecosystem dynamics of plant communities. It is applied to four National Hunting Reservoirs in the Catalan Pyrenees (Spain). In previous works several natural high-mountain ecosystems and population dynamics were modelled, but in those works grass was considered unlimited and the ecological succession of plant communities was not taken into account. In our new model we take advantage of the modularity of P systems, adding the plant communities to an existing model on scavengers dynamics [6]. We introduce the plant communities’ production and two possible changes or evolutions in communities: (1) due to less grazing pressure, and (2) due to the recovery of pastures with human management as for example fire or clearing. The simulator is defined using MeCoSim [12].

Keywords: Multi-environment, Probabilistic, Grasslands, Plant communities, Plant production, Plant succession.

1 Introduction

The knowledge on biological processes, their functioning and involved variables is increasing constantly. To define a model as close to reality as possible, it is helpful to use the newly acquired knowledge simultaneously in independent works. Like this the modelling of behavioural processes in different possible scenarios is enabled.

Natural processes are of great complexity, because each biological process involves a large number of variables and their interactions, so that modelling
ecosystems is not a simple task [5]. Classical models such as differential equations present several limitations. The most important one is the fact that they use a high computational cost to consider simultaneously multiple species and their interactions, whereas the very complex modelling makes them unviable. The most frequently used models are viability models or multi-agent models, that do not allow the study of different species with their interactions. In the work at hand, we make use of bio-inspired models, called P systems, which can serve as an alternative to the multi-agent models.

P systems are very useful to model natural ecosystems. Their properties make them very attractive for modelling complex ecosystems. Recently some works that model ecosystems using P Systems have been published [3, 4, 6]. In these works models focusing on animal population dynamics were presented. They model natural processes as feeding, grazing, reproduction and mortality. The food source is grass for ungulate species and meat or bones for avian scavengers.

In the present work, the amount of grass available for grazing is not a fixed value, as it was in previous ones. It depends on a large number of factors and furthermore on the existence of plant communities which evolve due to their management. The aim is to define new modules, which are incorporated into the model presented for the avian scavengers [6]. These modules operate in parallel with the modules of the existing scavengers model.

We have developed a model in which different high mountain plant communities (mostly alpine and subalpine communities) produce an amount of grass, in function of climate variables. In addition the plant communities can change and evolve over time due to changes in biotic or abiotic processes. Generally the abiotic processes occur in higher altitudes, and are not sufficiently known to be understood and modelled. The biotic processes play a more important role, as they cause changes in plant communities - in particular due to grazing management. Extreme environmental conditions in the highlands in winter - very low temperatures and a long-lasting snow cover - impede the growth of plants, so that only the growing season is modelled.

2 Materials and methods

The National Hunting Reserves of Catalonia (Spain) (NHR) are geographically defined territories with special characteristics, declared to promote, preserve and protect native fauna species. They are located in mountainous areas with high ecological and landscape quality. In these areas exists a very important wildlife fauna, including some species of great significance for hunting. In this work we have chosen the four National hunting reserves placed in the Catalan Pyrenees: Alt Pallars-Aran, Cerdanya-Alt Urgell, Cadí and Freser-Setcases (data provided by [11]). Over the centuries these alpine and subalpine zones have been exploited for agriculture and to feed sheep, cows and horses. This management has contributed to landscape transformation. The Pyrenean orography presents altitudinal zones between 1000 and more than 3000 m but our model only takes into
consideration the higher zones between 1700 and more than 3000 m, occupied mostly by grass, shrubland and conifer forest.

To model the high mountain ecosystem as correctly as possible, we have separated these zones into three different ranges, in function of their different climatic and orographic characteristics. As a result we have defined a low altitudinal zone - from 1700 to 2200 m - , an intermediate zone - from 2200 to 2600 m - and an upper zone - more than 2600 m. The first and the second range can be considered as low and high subalpine zones, while the third is alpine. This distinction is related to different plant communities available in each range, their primary production and the different length of their growing season, which is longer in the lower ranges.

Based on vegetation maps available [7], we define the areas corresponding to the four National Hunting Reserves (NHR), whereat areas with an altitude below 1700 meters are eliminated. As a result we get the entire area bounded by the NHR situated above 1700 meters, corresponding to subalpine and alpine biogeographic regions.

In the present work we have identified a total of 22 different phytosociology units, most of all alliances and, in few cases, classes\(^1\). Each plant alliance is associated to an average production and a standard deviation [2, 1, 8–10]. While the surface of the environment remains constant, the areas occupied by a given plant community can vary over time (secondary ecological succession).

The plant community production depends on the following considerations introduced into the model:

- Type of plant community; in this work we focus most of all on the alliance level.
- Weather, that determines the community’s production and the length of the growing season.
- Altitude, closely related to the previous concept. At higher altitudes temperatures are lower and the growing season is shorter.

The plant succession or evolution due to grazing and fire disturbances is analysed in our work: (1) the evolution due to less grazing pressure and, the inverse case, (2) the evolution due to the recovery of pastures with human management.

The beginning of the growing season is defined by means of the thermal integral. The period begins, for each altitudinal range, when the sum of positive daily temperatures, accumulated since the beginning of the year, exceeds 300 °C. The end of the period is set from the consulted references [10].

The animal species are sheep (\textit{Ovis aries}), cattle (\textit{Bos taurus}), horse (\textit{Equus caballus}), wild boar (\textit{Sus scrofa}) and pyrenean chamois (\textit{Rupicapra pyrenaica}).

\(^1\) As the case of classes is so rare, we will use from now on only the term ”alliances” referring to both cases.
3 Model

In order to model the ecosystem we use a multi-environment functional probabilistic P system with active membranes \([6]\) of degree \((5,4)\) (five membranes and four environments). We model the population dynamics of 5 animal species \((N = 5)\) and 22 plant communities \((NA = 22)\) in 4 different environments \((E = 4)\). We have 4 environments associated to each National Hunting Reservoir. A complete definition of the P system based framework was given in previous works \([4,6]\).

\[
(G, \Gamma, \Sigma, R_E, \Pi, \{r_{i,j} : r \in R_\Pi, 1 \leq j \leq 4\}, \{M_{i,j} : 0 \leq i \leq 4, 1 \leq j \leq 4\})
\]

Whereat:

1. The graph of the system is \(G = (\phi)\) because, in this case, there are no animal movements between environments.
2. The membrane structure is:

\[
\mu = [[1], [2], [3], [4]]
\]

The first three membranes are associated to the altitude: low, medium and high altitudinal ranges.

The initial configuration is:

\[
\text{Environment: } \{Ti, Ri\}
\]

\[
\text{Membranes: } M_0 = \{P_0, d_i, m : 1 \leq m \leq 3, 1 \leq i \leq N\},
\]

\[
M_i = \{X_{i,j}, A_w, U, \rho_0, \beta : 1 \leq i \leq N, 0 \leq j \leq g_i, 1 \leq v \leq NA\}
\]

3. The working alphabet of the P system is:

\[
\Gamma = \{X_{i,j}, Y_{i,j}, Z_{i,j}, Z'_{i,j}, W_{i,j,m} : 1 \leq i \leq N, 0 \leq j \leq g_i, 1 \leq k \leq E, 1 \leq m \leq 3\} \cup
\]

\[
\{d_i, m, d_i^m, a_i, e_i^m, d_i^m a_i, a_0, m : 1 \leq i \leq N, 1 \leq m \leq 3\} \cup
\]

\[
\{A_w, G_v, G_v', G_v'' : 1 \leq v \leq NA, 1 \leq m \leq 3\} \cup \{U, \beta, \lambda, \gamma\} \cup
\]

\[
\{P_0, \rho, \pi : 0 \leq i \leq 10\} \cup
\]

\[
\{T_m, R_m, N_{j,s} : 1 \leq j \leq 10, 1 \leq s \leq 10, 1 \leq m \leq 3\}
\]

Objects \(X_{i,j}, Y_{i,j}, Z_{i,j}, Z'_{i,j}, W_{i,j,m}\) represent the same animal but in different states. The objects \(d_i, m, d_i^m, a_i, e_i^m, d_i^m a_i, a_0, m\) and \(e'_{i,m}\) control the maximum number of animals per species in the ecosystem. If a species overcomes the maximum density values, it will be regulated. In all these objects index \(i\) is associated with the type of animal, index \(j\) is associated with the age, and \(m\) is the altitudinal range. \(A_w\) is a surface unit of the alliance \(v\) and \(G_v\) is the amount of grass produced per hectare by alliance \((A_v)\) and \(G_v'\) and \(G_v''\) indicate the total production in each altitudinal range \(m\). \(U\) is an object used to control the carrying capacity, and objects \(\beta, \gamma\) and \(\lambda\) are used to determine the grazing pressure level in the environment. \(T_m, R_m\) and \(R''_{m,j}\) are objects that include the climatic variability, \(T''\) for the length of the growing season and \(R''\) for the production of plant communities. The object \(N_{j,s}\) carries both information. At the end, objects \(P_i, \rho, \pi\) are counters that allow the synchronization of the P system. Necessary parameters introduced to model the plant communities’ dynamics, are given in table 1.

4. The environment alphabet is:

\[
\Sigma = \{R_i, T_i\} \cup \{T_k, R_k, T'_k, R'_k, : 1 \leq k \leq E, 1 \leq j \leq 10, 1 \leq s \leq 10\}
\]
$R_i, R_k, R'_k$ are objects that introduce the variability in plant production, while $T_i, T_k, T'_k$ are objects that include the variability in the length of the growing season.

5. Please contact the authors for more information on the model rules $R_E$ and $R_H$.

### Table 1. Parameters that affect animal and plant community dynamics (v animal species, $i$ plant community (alliance), $m$ altitudinal range, $k$ environment (NHR)).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Climatic variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random numbers</td>
<td>$1 \leq NZ \leq 100$</td>
<td>Animals</td>
<td></td>
</tr>
<tr>
<td>Animals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Animals population</td>
<td>$q_{v,j,m,k}, 1 \leq v \leq 5, 1 \leq j \leq g, 1 \leq m \leq 3, 1 \leq k \leq 4$</td>
<td>Mortality</td>
<td></td>
</tr>
<tr>
<td>Density regulation</td>
<td>$d_{v,m,k}, 1 \leq v \leq 5, 1 \leq m \leq 3, 1 \leq k \leq 4$</td>
<td>Hunting</td>
<td></td>
</tr>
<tr>
<td>Reproduction</td>
<td>$m_{v,k}, 1 \leq v \leq 5, 1 \leq k \leq 4$</td>
<td>Equivalent weight</td>
<td>$e_{v,m,k} \leq 5, 1 \leq v \leq 5$</td>
</tr>
<tr>
<td>Species age</td>
<td>$j_{v,m,k} \leq v \leq 5, 1 \leq j \leq g, 1 \leq m \leq 3, 1 \leq k \leq 4$</td>
<td>Plant communities</td>
<td></td>
</tr>
<tr>
<td>Reproduction</td>
<td>$r_{v,j,m,k} \leq v \leq 5, 1 \leq j \leq g, 1 \leq m \leq 3, 1 \leq k \leq 4$</td>
<td>Amount of grass produced per day (net primary production NPP)</td>
<td>$\mu_{R_i}, 1 \leq i \leq 22$</td>
</tr>
<tr>
<td>Standard deviation of plant net primary production</td>
<td>$\sigma_{R_i}, 1 \leq i \leq 22$</td>
<td>Mean of the growing season length</td>
<td>$\mu_{T_i}, 1 \leq i \leq 22$</td>
</tr>
<tr>
<td>Standard deviation of growing season length</td>
<td>$\sigma_{T_i}, 1 \leq i \leq 22$</td>
<td>Carrying capacity</td>
<td>$C_{ak}, 1 \leq k \leq 4$</td>
</tr>
<tr>
<td>Surface at low, medium and high altitude</td>
<td>$s_{v,k,m} \leq v \leq 5, 1 \leq k \leq 4$</td>
<td>Abandoned land evolution probability</td>
<td>$t_{a,i}, 1 \leq i \leq 22$</td>
</tr>
<tr>
<td>Fire evolution probabilities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fire probability per altitudinal range</td>
<td>$f_{a,i}, 1 \leq i \leq 22, 1 \leq m \leq 3$</td>
<td>Total surface (plant community, altitude range, environment)</td>
<td>$\delta_{i,m,k}, 1 \leq i \leq 22, 1 \leq m \leq 3, 1 \leq k \leq 4$</td>
</tr>
</tbody>
</table>

The model is structured into 8 modules, (see figure 1). In the following the different modules are described.

#### 3.1 Animal modules

The modules referring to the population dynamics (reproduction and mortality) were explained in detail in the Scavengers model [5, 6], therefore we only give a brief summary of all these modules in this work.

**Reproduction module** At the beginning an object of type X is associated with each animal. When rules from the reproduction module are applied to objects of type X, they evolve into objects of type Y. Objects associated with females that reproduce create new objects Y at age 0 ($Y_{i,0}$) and evolve to the object Z with the same index. The rules applied in this module are of the type: $[X_{i,j}]_h \xrightarrow{\alpha} [X_{i,j}, X_{i,0}]_h$.

**Mortality module** Two different mortality causes are considered: natural mortality and hunting mortality. When the domestic animals reach their life expectancy, they do not die, but they are removed from the ecosystem. Mortality rules are of the form:
3.2 Plant community modules

The following modules have been added to improve and complete the model of the ecosystem presented by Colomer et al. [6].

**Climate variability module** With the aim to introduce the climatic variability in the model, the objects $T_i$ and $R_i$ are used, whereat $T_i$ includes the variability of the duration of the growing season, and $R_i$ the communities’ production. Previously, we created a set of 10 random numbers following a normal distribution with mean 0 and standard deviation 1. In the environment labeled as 1, $T_i$ object evolves to four objects of type $T_{k,j}'$, and these are sent to their respective environments (2,3,4). The rest of $T_i$ objects, placed in the environments 2, 3 and 4 disappear at the same moment. The same occurs with the object $R_i$. $T_{k,j}'$ and $R_{k,s}'$ evolve in each environment into $T_{m,j}''$ and $R_{m,j}''$ and later into a new object $N_{j,s}$ in each altitudinal membrane, which contains the information of both objects $T_i$ and $R_i$ ($1 \leq j \leq 10, 1 \leq s \leq 10, 1 \leq k \leq E, 1 \leq m \leq 3$).

**Plant communities’ production module** The following rule produces an amount of grass, according to the information included into the object $N_{j,s}$.

\[
[Y_{t,g_{i,j}}|_{h} \xrightarrow{f_{c}} N_{j,s}|_{h}].
\]
Thus, when the object \( N_{j,s} \) and the object \( A_i \) (a surface unit of the alliance \( i \)) come together, they produce an amount of grass \( G_i \) available for herbivores and the object \( G'_i \), used to calculate the total grass production.

**Density regulation and grazing module** Once grass is produced, it serves as food for different species of ungulates present in the ecosystem. All animals need an average of food and choose a determined plant community for grazing. In this process it is possible to obtain two different scenarios: (1) the animal has enough available food and space \( a \), and (2) the animal cannot find resources. In the first case the animal (object \( Z \)) feeds, and is transformed into object \( W \). In the second case the movement module is applied.

**Movement module** In the case that the animals cannot find enough resources, they leave their zone (membrane \( m \)) with the following remaining set of objects \( (G \) and \( a) \) and go to membrane 4 of its environment. From there, objects that represent the animals can find resources and evolve into object \( W \), the rest of objects \( Z \) disappear.

**Plant communities’ evolution** Two different processes are introduced into the model encompassed within the plant communities’ evolution due to management: (1) less grazing pressure, and (2) recovering pastures with fire.

In the first case the object \( U \) controls the minimal carrying capacity required to maintain the type of grassland or plant community. If there is abandonment, the object \( \beta \) creates objects \( \gamma \) and \( \lambda \). After eight years without grazing \( (\lambda = 8) \), the alliance \( A_i \) can evolve into an alliance \( A_j \) with a stated probability. In the contrary case - with grazing - the object \( \beta \) disappears.

When the surface of the alliance \( A_j \) (scrubland) exceeds a certain value (50 ha), this plant community can evolve into grassland \( A_i \) with a stated probability due to human management (in this case fire).

**Updating module** This module aims to restore the initial configuration in order to start a new simulation.

## 4 Simulator and results

We define the simulator using MeCoSim [12]. The simulator incorporates two different types of menus: the fixed simulator menus, with which it is possible to carry out the simulation; and the specific menus, created to introduce the initial parameters of the ecosystem. The last one contains input, random numbers, parameter definitions, output and debug console. The input menu can be divided into several sub-menus depending on the needs.

The value of the parameters can be easily introduced and modified by the ecologist user in the table menus and hence it is possible to study the evolution of the ecosystem under different scenarios (see figures 2 and 3).
We have simulated the evolution of the plant communities and animals in 10 years, with 30 repetitions per year.

The results in plant communities’ production are displayed in figure 4, for each environment and for the first year of the simulation. The forest productions are not presented, because our animal species don’t use them. In this figure two variables are shown: the total grass production and the remaining grass production. The comparison of both variables indicates the amount of grass, expressed in kg of dry matter per ha and year, that is consumed by all ungulate species present in each environment during the seasonal period.

The plant communities’ evolution due to management is the second result shown in this work. Figure 5 provides a summary of the changing plant communities’ surface for three different years (1st, 5th and 10th year).

The most apparent change is produced in forest and shrubland surface, when human management (here: fire) transforms the woodland vegetation into grassland (Al. Bromion erecti).
The other change treated in this paper, the change due to less grazing pressure, is subtler. It can be observed only in some plant communities as for example in the *Nardion strictae* alliance or the *Bromion-Nardion* alliance in the environment N 2 (Cerdanya - Alt Urgell) see figure 5. This means that the carrying capacity is bigger than the real number of grazing ungulates. However this evolution from grass to woodland is partly masked by the reduction of wood communities.

5 Conclusions

In this paper we have presented a model to simulate the grassland dynamics based on P systems, which allows to simulate the behaviour under different scenarios. With this and previous works it is shown that a wide variety of natural processes can be successfully modelled by means of bio-inspired models. Furthermore P systems based models offer a modularity that makes them very useful.
This characteristic has permitted us to include our new plant communities modules into the Scavengers model [6] improving and completing it.

The results obtained after the first simulations show that the functioning of the model is adequate in all cases. These results follow the patterns observed in empirical experiences.

The next step is to improve and complete the model, incorporating more aspects: type of soil, slope and slope direction. Together they determine the vegetation development. Another interesting point to incorporate into the model is to expand the extent of possible evolution of plant communities.

References

Fig. 5. Changes in plant communities composition along 10 years.


vised Selected Papers, pages 144–159. Springer-Verlag, August 2010.
List of Authors

Adorna, H., 77, 225
Agrigoroaiei, O., 53
Alhazov, A., 3, 63, 471
Beal, J., 15
Bodenstein, C., 191
Burtseva, L., 471
Cabarle, F., 77
Cienciala, L., 93
Ciencialová, L., 93
Ciobanu, A., 479
Ciobanu, G., 53
Cojocaru, S., 471
Colesnicov, A., 471
Colomer, M. A., 485, 497
Cuihaj-Varjú, E., 109
Dinneen, M.J., 127
Dittrich, P., 169
Escuela, G., 169
Fillat, F., 497
Fondevilla, C., 485, 497
Frisco, P., 151
García-Quismondo, M., 323
Garg, D., 381
Gheorghe, M., I
Giavitto, J.-L., 17
Gómez, G., 153
Gruenert, G., 169
Heiland, I., 191
Hinze, T., 169, 191
Hoogeboom, H.J., 299
Huang, X., 213
Ibo, G. N., 225
Ipate, F., 213, 241, 459, 479
Ivanov, S., 253
Jebasingh, S., 271
Kefalas, P., 421
Kim, Y.-B., 127
Krasovitskiy, A., 63
Krithivasan, K., 283, 381
Langer, M., 93
Lefîtcaru, R., 241
Leporati, A., 433
Lombardo, R., 347
Macías-Ramos, L. F., 323
Macari, Y., 315
Magariu, G., 315
Malahov, L., 471
Manca, V., 347, 365
Marchetti, L., 365
Martínez-del-Amor, M.A., 77
Mauri, G., 433
Metta, V.P., 381
Montori, A., 485
Nagar, A.K., 271
Nicolescu, R., 33, 127
Obtulowicz, A., 395
Pérez-Hurtado, I., 323
Pérez-Jiménez, M.J., 323
Páun, A., 409
Páun, G., I
Pan, L., 299
Petreska, I., 421
Porreca, A.E., 433
Pérez-Hurtado, I., 241
Pérez-Jiménez, M. J., 241
Ramanujan, A., 283
Riscos-Núñez, A., 323
Robinson, T., 271
Rogozhin, Y., 63, 471
Shburlan, D., 447
Schau, B., 191
Schuster, S., 191
Ţurcanu, A., 459
Tudose, C., 241
Valencia-Cabrera, L., 323
Valencia-Cabrera, L., 497
Vaszil, G., 43, 109
Verlan, S., I
Verlan, T., 315
Zandron, C., 433
Zhang, G., 213