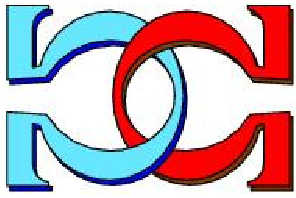
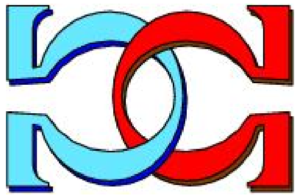
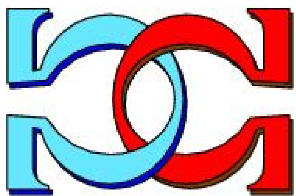
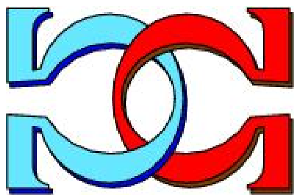


CDMTCS
Research
Report
Series

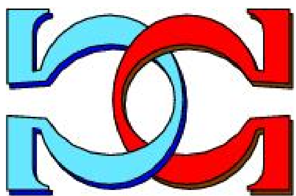




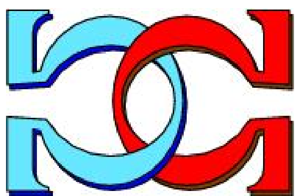
Parallel and Distributed Algorithms in P Systems



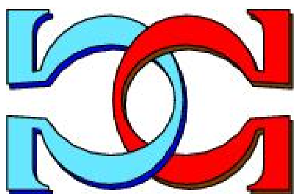
Radu Nicolescu
Department of Computer Science,
University of Auckland,
Auckland, New Zealand



CDMTCS-415
December 2011



Centre for Discrete Mathematics and
Theoretical Computer Science



Parallel and Distributed Algorithms in P Systems

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

Abstract

Our group's recent quest has been to use P systems to model parallel and distributed algorithms. Several framework extensions are recalled or detailed, in particular, modular composition with information hiding, complex symbols, generic rules, reified cell IDs, asynchronous operational modes, asynchronous complexity. We motivate our proposals via P system models of several well-known distributed algorithms, such as leader election and distributed echo. As another type of application, we mention a dynamic programming algorithm for stereo matching in image processing. We suggest criteria to assess the merits of this modelling approach and offer preliminary evaluations of our proposed additional ingredients, which have been useful in refactoring existing systems and could be useful to the larger P systems community.

Keywords: P systems, P modules, complex symbols, generic rules, cell IDs, distributed algorithms, parallel algorithms, synchronous networks, asynchronous networks, leader election, distributed echo, 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 [29]. An in-depth overview of this model can be found in Păun et al. [31].

Broadly speaking, typical P system research falls into one of the following three areas, which could be labelled: (1) *theory*, such as computational completeness (universality), complexity classes (e.g., polynomial solutions to NP-hard problems) or relationships with other models (e.g., automata, grammar systems and formal languages); (2) *tools*, including designers, simulators and verifiers; and (3) *applications*, most of these are in computational biology, but also in a large variety of other areas, such as biomedicine, economics or linguistics. For a more comprehensive list, refer the to Păun et al.'s survey [31].

This talk discusses our group's recent use of P systems to model *parallel and distributed algorithms*, a complementary application area which, except for a few notable exceptions, such as the pioneering work of Ciobanu et al. [8, 9], has not been addressed by other research. We intentionally use a *maximalist* approach, by selecting the most adequate ingredients for our quest and propose several extensions, which seem to "naturally" fit into the existing P systems framework and are useful or even *required* for modelling fundamental distributed and parallel algorithms.

To interest us, a distributed problem should score high on the following criteria:

- The problem must be "self-referential", i.e. the problem should be given by the P system itself, more specifically, by its topology, and not as externally encoded data, which is then fed into a different P system.

- The problem must be a fundamental or very challenging distributed problem.

- The P solution (the P rules which solve the problem) must be short and crisp, comparable to the best existing pseudo-code, which further implies that the number of rules must be *fixed*, regardless of the scale of the problem instance.

- The P solution must be comparable in efficiency to the best known algorithms, i.e. the number of P steps (a P step is a single time unit in P systems) must be comparable to the number of steps or rounds.

In this spirit, we have studied a variety of topics:

- *Hyperdag P systems (hP)*: DAG or hypergraph based models, which seem more adequate for well structured scenarios where the tree model is inadequate, such as computer networks or phylogenetic trees enhanced with horizontal gene transfers. Interestingly, hP systems are in general not planar, but admit a Moebius-like graphical representation.

- *Network discovery*: algorithms to discover and search the neighbourhood and the whole digraph—in particular, several algorithms to establish disjoint paths.

- *FSSP*: several variants of the firing squad synchronization problem (which should probably be called simultaneous neuron firing).

- *Fault tolerant distributed computing*: Byzantine agreement (the "crown jewel" of distributed algorithms).

- *P modules*: a model which favours recursive composition with information hiding.

- *Asynchronous P systems*: a recent proposal for asynchronous P systems, more closely related to the mainstream concepts in distributed algorithms, extended with a validation suite consisting of several efficient asynchronous distributed DFS and BFS algorithms.

- *Parallel stereo matching*: a recent design of a massively parallel image processing task.

Here, I report our design choices which have proved most useful and have been refined in the process. We hope that, by looking back at parallel and distributing systems (which were one the inspiration sources for the P systems framework), our experience will be useful to the global P systems community, for developing complex modelling applications.

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 in most of our recent papers [1, 10, 12, 13, 11, 14, 15, 16, 21, 25, 26, 27], our preferred membrane structure is a *directed graph* (*digraph*) or one of its subclasses, such as a *directed acyclic graph* (*DAG*) or, occasionally, a (rooted) *tree*, or a more complex structure, such as a *hypergraph* or a *multigraph*; (undirected) *graph* structures can be emulated by symmetric digraphs. Arcs represent *duplex* channels, so parents (arc tails) can send messages to children (arc heads) *and* children can send messages to parents, i.e. messages can travel along both forward or reverse arcs' directions.

Each arc has *two* labels—one at its tail and another at its head: note that this is an extension of the usual graph convention, where an arc has just one label. Arc labels can be used for directing messages over a specific arc. Labels can be explicitly indicated: otherwise, we assume a default labelling convention. A *not* explicitly labelled arc, $\alpha = (\sigma_i, \sigma_j)$, is implicitly labelled with the indices of its two adjacent cells: j—on its tail and i—on its head. Figure 4 shows an explicitly labelled arc, (σ_1, σ_2), labelled as in the default case.

In the basic model, all cells evolve *synchronously*. Rules are prioritized (i.e. linearly ordered) and applied in *weak priority* order [31]. The general form of a rule [15, 27], which transforms state S to state S', is

$$S\ x \rightarrow_{\alpha} S'\ x'\ (y)_{\beta} \ldots\ |_{z},$$

where:

- S, S' are states, $S, S' \in Q_i$;
- x, x', y, z are strings which represent multisets of symbols, z being a *promoter*, $x, x', y, z \in O^*$;
- α is a *rewriting* operator, $\alpha \in \{\texttt{min}, \texttt{max}\}$;
- β is a *transfer* operator, $\beta \in \{\uparrow_{\gamma}, \downarrow_{\gamma}, \updownarrow_{\gamma} \mid \gamma \in \{\forall, \exists\} \cup \Lambda\}$, where $\gamma = \forall$ is the default and Λ is the set of (implicit or explicit) arc labels.

The transfer operator β's arrow points in the direction of transfer: $\uparrow$—towards parents; $\downarrow$—towards children; $\updownarrow$—in both directions. Note that:

- By default, we consider *duplex* (bidirectional) communications.
- If all rules exclusively use $\downarrow$, then our system use *simplex* (unidirectional) communications, from structural parents to structural children.
- If all rules exclusively use $\updownarrow$ arrows, then arc directions do not matter: this offers one way to consider (undirected) *graphs*, at the rule level (another way, at the structural level, is to consider *symmetric* digraphs).

The transfer operator β's qualifier, γ, indicates the distribution form: $\forall$—a broadcast (the default); $\exists$—an anycast (nondeterministic); or 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 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 [25]. This is a strong requirement; it 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, independent of how many cells are included in the system.

3 Extensions—P modules

The above definition corresponds to what we called a *simple P module* [11]. Figuratively, if we allow *half-arcs* (i.e. arc "stumps" or disconnected arc heads and tails), we obtain more general *P modules* [12]. The open ends define *ports*, through which P modules transfer messages. P modules define a controlled way ro *recursively compose* P systems, where internal features of one P module are *hidden* and inaccessible to other P modules.

Without going into detail (available in [12]), I illustrate these concepts by a series of figures, where dotted lines delimit P modules. Figure 1 illustrates this idea in a very simplistic scenario. Figure 2 illustrates the recursive modular composition of a P module which solves the greatest common divisor (GCD) problem. Figure 3 illustrates the use of P modules to systematicaly build proper connections in a Byzantine agreement scenario.

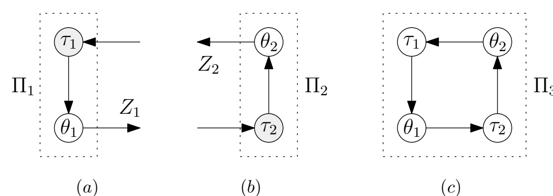


Figure 1: A simple composition of two P modules.

4 Extension—Complex Symbols

While atomic symbols are sufficient for many theoretical studies (such as computational completeness), complex algorithms need appropriate data structures. Previous studies

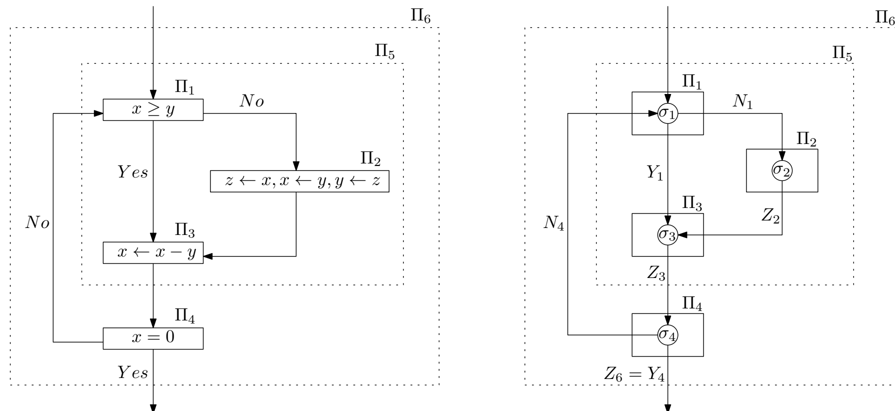


Figure 2: Left: A flowchart for computing the GCD. Right: A corresponding P module built by recursive modular composition.

have proposed complex symbols in the form of strings. While string symbols are enough in some scenarios, they are not adequate for complex algorithms, because strings require complex and costly encoding and decoding (parsing), which would adversely clutter the algorithm's description and affect its runtime performance. We proposed a simple form of complex symbols, similar to Prolog terms and Lips tuples, with a crisp and fast encoding and decoding and simplified "unification" semantics [27]. Such complex symbols can be viewed as complex molecules, consisting of elementary atoms or other molecules.

We thus enhance our initial vocabulary, by recursive composition of *elementary symbols* from O into *complex symbols*, which are compound terms of the form:

$$t(i, \dots),$$

where

- t is an *elementary symbol* representing the functor;
- i can be
  - an *elementary* symbol,
  - another *complex* symbol,
  - a *free variable* (open to be bound, according to the cell's current configuration), or
  - (in more complex scenarios) a *multiset* of elementary and complex symbols and free variables.

Also, we often abbreviate complex symbols (i) by using subscripts for term arguments and (ii) by using superscripts for runs of repeated occurrences of the same functor. The following are examples of such complex symbols, where a, b, c, d, e, f are elementary symbols and i, j are free variables (assuming that these are not listed among elementary symbols):

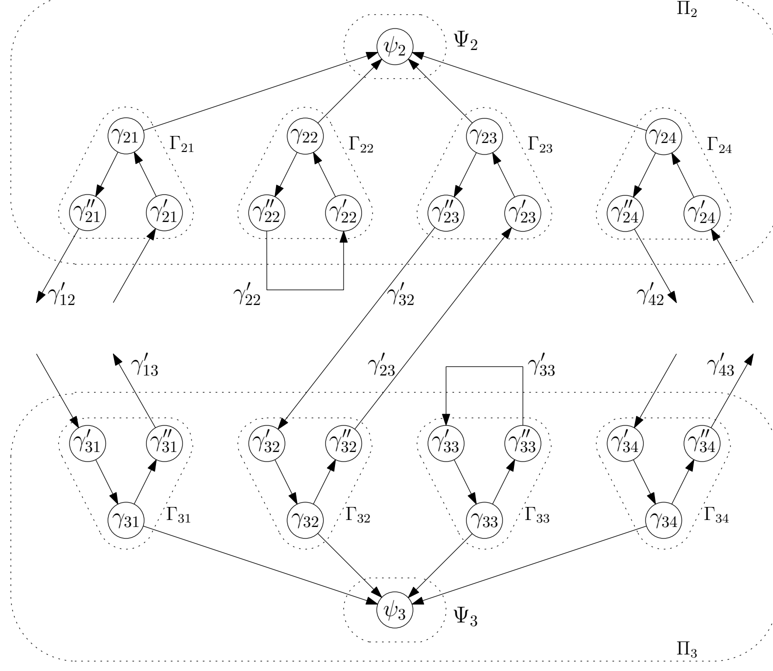


Figure 3: Modular design a Byzantine agreement scenario.

- $b(2) = b_2$,
- $c(i) = c_i$,
- $d(i, j) = d_{i,j}$,
- $c(a^2b^3)$,
- $d(e, i, f(j))$,
- $c(), c(c()) = c^2(), c(c(c())) = c^3(), \ldots,$

Note that the sequence items $c(), c^2(), c^3(), \ldots$ can be interpreted as integers, $0, 1, 2, \ldots$ (with an excess functor, to smooth the treatment of 0).

We use complex symbols as required: (i) for cell contents (including promoters), (ii) for cell IDs (described in the next section), (iii) for states (which can be viewed as symbols with specific semantics) and (iv) for arc labels.

5 Extension—Generic Rules

Further, we process our multisets of complex symbols with high-level *generic rules*, using *free variable* matching. This approach is a practical necessity, because:

1. it enables reasonably fast parsing and processing of subcomponents (practically impossible with string symbols) and

2. it allows us to describe an algorithm with a *fixed size elementary alphabet* and a *fixed sized rule set*, independent of the number of cells in the system (sometimes impossible with only atomic symbols).

A generic rule is identified by using complex symbols and an extended version of the classical rewriting mode, in fact, a combined *instantiation.rewriting* mode, which is one of `min.min`, `min.max`, `max.min`, `max.max`, i.e. we consider all possible combinations between (1) an *instantiation* mode in $\mathtt{min}, \mathtt{max}$ and (2) a *rewriting* mode in $\mathtt{min}, \mathtt{max}$.

To explain generics, consider a cell, σ, containing three counters, $c^2()$, $c^2()$, $c^3()$, (respectively interpreted as numbers $1, 1, 2$), and all four possible instantiation.rewriting modes of the following "decrementing" rule:

$$(\rho_\alpha)\ S_1\ c^2(i) \rightarrow_\alpha S_2\ c(i).$$

where $\alpha \in \{\mathtt{min.min}, \mathtt{min.max}, \mathtt{max.min}, \mathtt{max.max}\}$.

1. If $\alpha = \mathtt{min.min}$, rule $\rho_{\mathtt{min.min}}$ nondeterministically generates *one* of the following rule instances:

$$\begin{array}{ll} (\rho_1') & S_1\ c^2() \rightarrow_{\mathtt{min}} S_2\ c() \quad \text{or} \\ (\rho_1'') & S_1\ c^3() \rightarrow_{\mathtt{min}} S_2\ c^2(). \end{array}$$

   In the first case, using (ρ_1'), cell σ ends with counters $c()$, $c^2()$, $c^3()$, i.e. numbers $0, 1, 2$. In the second case, using (ρ_1''), cell σ ends with counters $c^2()$, $c^2()$, $c^2()$, i.e. numbers $1, 1, 1$.

2. If $\alpha = \mathtt{max.min}$, rule $\rho_{\mathtt{max.min}}$ generates *both* following rule instances:

$$\begin{array}{ll} (\rho_2') & S_1\ c^2() \rightarrow_{\mathtt{min}} S_2\ c() \quad \text{and} \\ (\rho_2'') & S_1\ c^3() \rightarrow_{\mathtt{min}} S_2\ c^2(). \end{array}$$

   In this case, using (ρ_2') and (ρ_2''), cell σ ends with counters $c()$, $c^2()$, $c^2()$, i.e. numbers $0, 1, 1$.

3. If $\alpha = \mathtt{min.max}$, rule $\rho_{\mathtt{min.max}}$ nondeterministically generates *one* of the following rule instances:

$$\begin{array}{ll} (\rho_3') & S_1\ c^2() \rightarrow_{\mathtt{max}} S_2\ c() \quad \text{or} \\ (\rho_3'') & S_1\ c^3() \rightarrow_{\mathtt{max}} S_2\ c^2(). \end{array}$$

   In the first case, using (ρ_3'), cell σ ends with counters $c()$, $c()$, $c^3()$, i.e. numbers $0, 0, 2$. In the second case, using (ρ_3''), cell σ ends with counters $c^2()$, $c^2()$, $c^2()$, i.e. numbers $1, 1, 1$.

4. If $\alpha = \mathtt{max.max}$, rule $\rho_{\mathtt{min.max}}$ generates *both* following rule instances:

$$\begin{array}{ll} (\rho_4') & S_1\ c^2() \rightarrow_{\mathtt{max}} S_2\ c() \quad \text{and} \\ (\rho_4'') & S_1\ c^3() \rightarrow_{\mathtt{max}} S_2\ c^2(). \end{array}$$

   In this case, using (ρ_4') and (ρ_4''), cell σ ends with counters $c()$, $c()$, $c^2()$, i.e. numbers $0, 0, 1$.

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.

For all modes, the instantiations are *ephemeral*, created when rules are tested for applicability and disappearing at the end of the step. Further examples appear in the next section.

6 Extension—Cell IDs

The well-known *distributed leader election* problem and its famous impossibility result, as presented by Lynch [23] and Tel [32], highlights the need for *reified cell IDs*[1]. Leader election is a fundamental problem in distributed algorithms and can be viewed as a highly abstract and simplified version of cellular differentiation in developmental biology. Imagine a network of cells which must elect a leader. A celebrated result shows that this is *impossible*, in the *deterministic* case, 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, σ_i, start with its own unique *cell ID* symbol, ι_i, where ι is a dedicated cell ID functor. We thus *reify* the external cell index i into an internal complex symbol, which is accessible to the rules; in fact, we will use it exclusively as an *immutable promoter* [27].

Figure 4 shows a ring structured system, where cells contain cell ID symbols, ι_i; this breaks the symmetry and enables the leader election process.

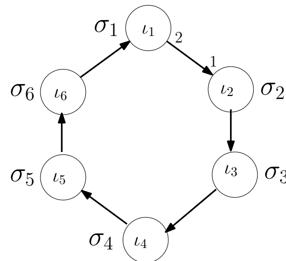


Figure 4: A ring structured system, where cells contains cell ID symbols (σ_1 contains $\iota_1, \ldots$); note also that arc (σ_1, σ_2) is explicitly labelled with its default labels.

Note that, as suggested by the counters discussed in the previous section, cell IDs do *not* need to increase the alphabet size— we can encode any number of cell IDs with a fixed number of elementary symbols, e.g., as unary or binary strings.

To explain cell IDs, consider this *generic* rule:

$$S_3\ a\ n_j \rightarrow_{\texttt{min.min}} S_4\ b\ (c_i)_{\updownarrow_j}\ |\iota_i.$$

[1] *To reify* = to consider or make (an abstract idea or concept) real or concrete; cf. republic = res publica (Lat.) = public object.

This generic rule uses the `min.min` instantiation.rewriting mode and complex symbols, c_i and n_j, where i and j are free variables (recall that c_i and n_j are shorthands for $c(i)$ and $n(j)$).

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, ι_i;
2. j—because it also indicates the target of the transfer mode, $\updownarrow_j$, if we assume that arcs are implicitly labelled by cell indices.

Briefly:

1. according to the *first* `min`, this rule is *instantiated once*, for one of the existing n_j symbols (if any), while promoter, ι_i, constrains i to the cell ID index of the current cell, σ_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 neighbour j (if this neighbour exists, as parent or child).

As a more elaborate example, consider a system with N cells, σ_1, σ_2, …, σ_N, where cell σ_1 has two structural neighbours, σ_2 and σ_3, is in state S_3 and contains multiset $a^2 n_2^2 n_3$. Consider also all four possible instantiations of the following rule, ρ_α, where α is one of the four extended rewriting modes:

$$(\rho_\alpha)\ S_3\ a\ n_j \rightarrow_\alpha S_4\ b\ (c_i)_{\updownarrow_j}\ |\iota_i.$$

- Rule $\rho_{\texttt{min.min}}$ generates one of the two low-level instances:
  either $S_3\ a\ n_2 \rightarrow_{\texttt{min}} S_4\ b\ (c_1)_{\updownarrow_2}$ or $S_3\ a\ n_3 \rightarrow_{\texttt{min}} S_4\ b\ (c_1)_{\updownarrow_3}$.
- Rule $\rho_{\texttt{min.max}}$ generates one of the two low-level instances:
  either $S_3\ a\ n_2 \rightarrow_{\texttt{max}} S_4\ b\ (c_1)_{\updownarrow_2}$ or $S_3\ a\ n_3 \rightarrow_{\texttt{max}} S_4\ b\ (c_1)_{\updownarrow_3}$.
- Rule $\rho_{\texttt{max.min}}$ generates the two low-level instances:
  $S_3\ a\ n_2 \rightarrow_{\texttt{min}} S_4\ b\ (c_1)_{\updownarrow_2}$ and $S_3\ a\ n_3 \rightarrow_{\texttt{min}} S_4\ b\ (c_1)_{\updownarrow_3}$.
- Rule $\rho_{\texttt{max.max}}$ generates the two low-level instances:
  $S_3\ a\ n_2 \rightarrow_{\texttt{max}} S_4\ b\ (c_1)_{\updownarrow_2}$ and $S_3\ a\ n_3 \rightarrow_{\texttt{max}} S_4\ b\ (c_1)_{\updownarrow_3}$.

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 $\rho_{\texttt{max.max}}$, cell σ_1 would need its own custom rule set, consisting of N low-level rules:

$$\{S_3\ a\ n_j \rightarrow_{\texttt{max}} S_4\ b\ (c_1)_{\updownarrow_j} \mid 1 \le j \le N\}.$$

We argue that our approach has positive consequences, both at the conceptual and practical (implementation) level. In fact, this was critical to achieve our goal of solving complex distributed problems with *fixed* size rule sets (independent of the problem size).

7 Asynchronous P Systems

The traditional P system model is *synchronous*, i.e. all cells evolve controlled by a single global clock. P systems with various *asynchronous* features have been recent investigated [18, 3, 7, 4, 6, 5, 17, 22, 28, 33]. We are looking for similar but simpler definitions, closer to the standard definitions used in distributed algorithms [23, 32]. We are interested to model fundamental and challenging asynchronous distributed algorithms and to assess the merits of such modelling exercise.

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 evolution.

Our approach, based on classical notions in distributed algorithms [32], 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):

- For each cell, each step starts after a *random step delay*, t, after the preceding step.
- For each cell, its rules application step, once started, takes *zero* time (it occurs instantaneously).
    - Note that a small execution delay might look more realistic, but could needlessly complicate the arguments.
- For each message, its *delivery delay*, t, is *random*:
    - either from its origin;
    - or, more realistically, after the previous message sent over the same channel (arc).
- We typically assume that messages sent over the same arc arrive in strict *queue* order (FIFO)—but one could also consider arrival in *arbitrary* order (bag, instead of queue).
- The message granularity is still an open question. Should a message contain: (a) A single symbol (elementary or complex)? (b) All symbols sent by the same rule? (c) All symbols sent from a cell, during an application step?

Note that classical 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.

This proposal suggests further directions of study of fundamental notions and properties related to asynchronous systems [23, 32], such as causality, liveness, safety, fairness and specific proof and verification techniques.

We next present two more elaborate examples, described in more detail elsewhere [1]: an algorithm for neighbours discovery and a version of the well-known Echo algorithm [32].

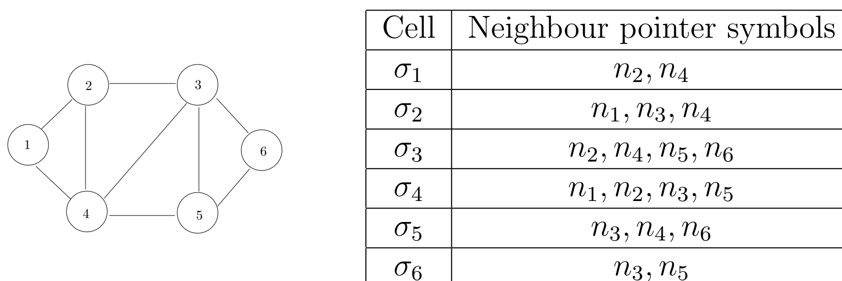
8 Discovering Neighbours (Async)

Many distributed P algorithms require that cells are aware of their local topology, i.e. each cells knows its neighbours' IDs.

In this algorithm, all cells start in the same initial state, S_0, with the same set of rules. Each cell, σ_i, contains a cell ID symbol, ι_i, which is *immutable* and used as a *promoter*. Additionally, the source cell, σ_s, is marked with one symbol, a. All cells end in the same state, S_3. On completion, each cell contains its cell ID symbol, ι_i, and neighbour pointers, n_j. The source cell, σ_s, is still marked with symbol a.

Figure 5 illustrates a sample graph and the discovered neighbourhoods. Listing 6 presents the solution, i.e. the rule set which solves this problem. Note that rule 1.1 is generic and uses a cell ID promoter, which was instrumental in ensuring a fixed size rule set. For more details, refer to [1].

Although the algorithm terminates, cells involved in this algorithm have no way of knowing when the algorithm terminates. This problem is shared by other asynchronous algorithms and highlights some of the difficulties faced by the asynchronous model: for details and possible solutions refer to [32].



Cell	Neighbour pointer symbols
σ_1	n_2, n_4
σ_2	n_1, n_3, n_4
σ_3	n_2, n_4, n_5, n_6
σ_4	n_1, n_2, n_3, n_5
σ_5	n_3, n_4, n_6
σ_6	n_3, n_5

Figure 5: A sample graph and the discovered neighbourhoods.

9 Echo Algorithm (Async)

The Echo algorithm is a *wave* algorithm [32]. It starts from a source cell, which broadcasts forward messages. These forward messages transitively reach all cells and, at the end, are reflected back to the initial source. The forward phase establishes a *virtual spanning tree* and the return phase is supposed to follow up its branches. The tree is virtual, because it does not involve any structural change: instead, virtual child-parent links are established

0. Rules for state S_0:

   1 $S_0\ a \to_{\texttt{min}} S_1\ ay\ (z) \updownarrow_{\forall}$

   2 $S_0\ z \to_{\texttt{min}} S_1\ y\ (z) \updownarrow_{\forall}$

   3 $S_0\ z \to_{\texttt{max}} S_1$

1. Rules for state S_1:

   1 $S_1\ y \to_{\texttt{min.min}} S_2\ (n_i) \updownarrow |_{\iota_i}$

   2 $S_1\ z \to_{\texttt{max}} S_2$

2. Rules for state S_2:

   1 $S_2 \to_{\texttt{min}} S_3$

   2 $S_2\ z \to_{\texttt{max}} S_3$

Figure 6: Rules for discovering neighbourhoods.

by pointer symbols. The algorithm terminates when the source cell receives the reflected messages from all its neighbours.

This algorithm requires that each cell "knows" all its neighbours (structural heads and tails). This could be realised by a preliminary phase which builds this knowledge, such as presented in Section 8. Therefore, we assume that each cell already knows all neighbours' IDs.

Scenario 1 in Figure 7 assumes that all messages arrive in one time unit, i.e. in *synchronous* mode. The forward and return phases take the same time, i.e. D time units each, where D is the diameter of the underlying graph, G. Scenario 2 in Figure 8 assumes that some messages travel much faster than others, which is possible in *asynchronous* mode: $t = \epsilon$, where $0 < \epsilon \ll 1$. In this case, the forward and return phases take very different times: D and $N - 1$ time units, respectively, where N is the number of nodes of graph G. At first sight, this seems paradoxical and highlights some of the subtleties involving runtime estimates for asynchronous algorithms. For more details, including the rule set (omitted here), refer to our paper [1].

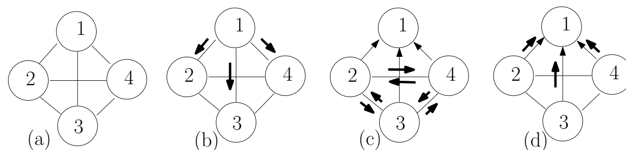


Figure 7: The Echo algorithm in synchronous mode. Edges with arrows indicate child-parent arcs in the virtual spanning tree built by the algorithm. Thick arrows near edges indicate messages. Steps (b), (c), (d) take one time unit each.

10 Parallel Stereo Matching

Image processing offers many opportunities for parallel modelling, but, with a few notable exceptions, mostly from the Seville group [2, 30], has not yet attracted much attention from the P systems community.

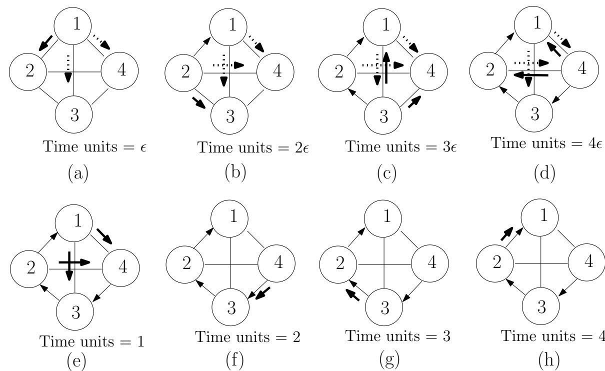


Figure 8: One possible evolution of the Echo algorithm in asynchronous mode, with different forward and return times. Dotted thick arrows near edges indicate messages still in transit. Steps (a), (b), (c), (d) take ϵ time units each; steps (e), (f), (g), (h) take one time unit each.

To finalize, 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.

This modelling exercise has enabled us to *generalise* and *refactor* our matching algorithm, 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 mentioned in this paper, such as labelled multigraph structures and generic rules with complex symbols.

Figure 9 shows, in order, a monocular left image, a monocular right image and their true disparity map (the ground truth). This is the well-known Tsukuba head-and-lamp stereo pair, first proposed by Nakamura et al. [24]. Figure 10 shows our computed disparity maps: left—by the old program; right—by our refactored and better tuned program.

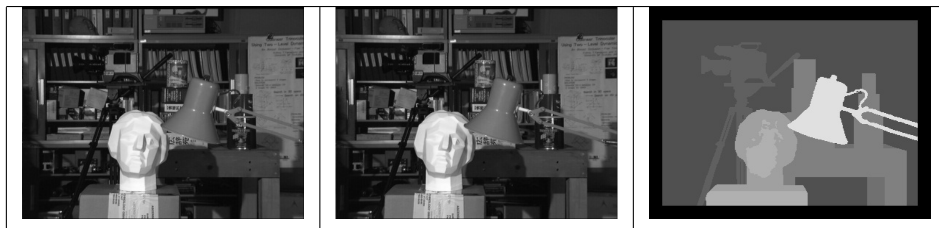


Figure 9: In order: monocular left image, monocular right image, true disparity map.

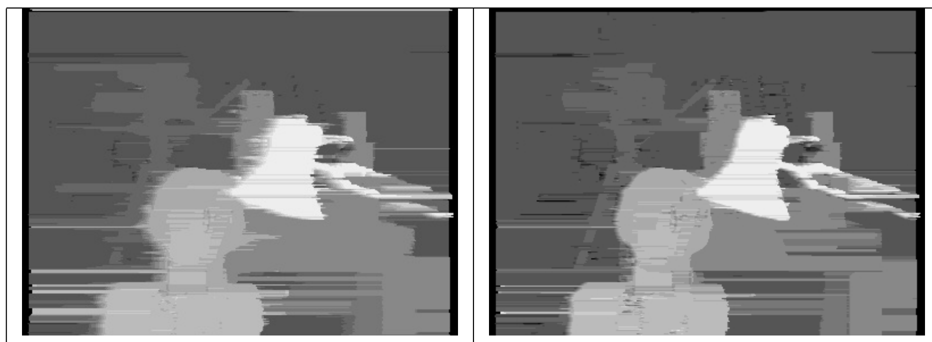


Figure 10: Computed disparity maps: left—old program; right—refactored program.

11 Conclusions

We suggested a few simple criteria for assessing the merits of using P systems for modelling complex parallel and distributed algorithms, such as a fixed set of short and crisp rules and a runtime performance similar to state-of-art pseudo-code. We discussed a few P system ingredients which are useful or even required to achieve these goals, such as complex symbols, generic rules, cell IDs, provisions for modular development and asynchronous processing. We note that some of these specific details need further study. We believe that our proposed additional ingredients have proved their usefulness in refactoring existing realistic systems and could be interesting to the larger P systems 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

[1] T. Bălănescu, R. Nicolescu, and H. Wu. Asynchronous P systems. *International Journal of Natural Computing Research*, pages 1–18, in press, 2011.

[2] J. Carnero, D. Díaz-Pernil, H. Molina-Abril, and P. Real. Image segmentation inspired by cellular models using hardware programming. In R. González-Díaz and P. Real-Jurado, editors, *3rd International Workshop on Computational Topology in Image Context*, pages 143–150, 2010.

[3] G. Casiraghi, C. Ferretti, A. Gallini, and G. Mauri. A membrane computing system mapped on an asynchronous, distributed computational environment. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 159–164. Springer, 2005.

[4] M. Cavaliere, O. Egecioglu, O. Ibarra, M. Ionescu, G. Păun, and S. Woodworth. Asynchronous spiking neural P systems: Decidability and undecidability. In M. Garzon and H. Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 246–255. Springer Berlin / Heidelberg, 2008.

[5] M. Cavaliere, O. H. Ibarra, G. Păun, O. Egecioglu, M. Ionescu, and S. Woodworth. Asynchronous spiking neural P systems. *Theor. Comput. Sci.*, 410:2352–2364, May 2009.

[6] M. Cavaliere and I. Mura. Experiments on the reliability of stochastic spiking neural P systems. *Natural Computing*, 7:453–470, December 2008.

[7] M. Cavaliere and D. Sburlan. Time and synchronization in membrane systems. *Fundam. Inf.*, 64:65–77, July 2004.

[8] G. Ciobanu. Distributed algorithms over communicating membrane systems. *Biosystems*, 70(2):123–133, 2003.

[9] G. Ciobanu, R. Desai, and A. Kumar. Membrane systems and distributed computing. In G. Păun, G. Rozenberg, A. Salomaa, and C. Zandron, editors, *WMC-CdeA*, volume 2597 of *Lecture Notes in Computer Science*, pages 187–202. Springer-Verlag, 2002.

[10] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. Toward practical P systems for distributed computing. In D. Enachescu and R. Gramatovici, editors, *Seria Matematica-Informatica (Honor of Solomon Marcus on the Occasion of his 85th Anniversary)*, pages 23–34. Analele Universitatii Bucuresti, 2009.

[11] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. Edge- and node-disjoint paths in P systems. *Electronic Proceedings in Theoretical Computer Science*, 40:121–141, 2010.

[12] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. A faster P solution for the Byzantine agreement problem. In M. Gheorghe, T. Hinze, and G. Păun, editors, *Conference on Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 175–197. Springer-Verlag, Berlin Heidelberg, 2010.

[13] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. P systems and the Byzantine agreement. *Journal of Logic and Algebraic Programming*, 79(6):334–349, 2010.

[14] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. Synchronization in P modules. In C. S. Calude, M. Hagiya, K. Morita, G. Rozenberg, and J. Timmis, editors, *Unconventional Computation*, volume 6079 of *Lecture Notes in Computer Science*, pages 32–44. Springer-Verlag, Berlin Heidelberg, 2010.

[15] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. An adaptive algorithm for P system synchronization. In *Twelfth International Conference on Membrane Computing (CMC12), Fontainebleau/Paris, France, August 23-26, 2011, Proceedings*, pages 127–152, 2011.

[16] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. Faster synchronization in P systems. *Natural Computing*, pages 1–9, 2011.

[17] R. Freund. Asynchronous P systems and P systems working in the sequential mode. In *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 36–62. Springer Berlin / Heidelberg, 2005.

[18] P. Frisco. The conformon-P system: a molecular and cell biology-inspired computability model. *Theor. Comput. Sci.*, 312:295–319, January 2004.

[19] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[20] G. L. Gimel'farb. Probabilistic regularisation and symmetry in binocular dynamic programming stereo. *Pattern Recognition Letters*, 23(4):431–442, 2002.

[21] G. L. Gimel'farb, R. Nicolescu, and S. Ragavan. P systems in stereo matching. In A. Berciano, editor, *Conference on Computer Analysis of Images and Patterns*, volume 6855 of *Lecture Notes in Computer Science*, pages 285–292. Springer-Verlag, Berlin Heidelberg, 2011.

[22] J. Kleijn and M. Koutny. Synchrony and asynchrony in membrane systems. In *Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 66–85. Springer Berlin / Heidelberg, 2006.

[23] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[24] Y. Nakamura, T. Matsuura, K. Satoh, and Y. Ohta. Occlusion detectable stereo—occlusion patterns in camera matrix. In *CVPR*, pages 371–378. IEEE Computer Society, 1996.

[25] R. Nicolescu, M. J. Dinneen, and Y.-B. Kim. Discovering the membrane topology of hyperdag P systems. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 410–435. Springer-Verlag, 2009.

[26] R. Nicolescu, M. J. Dinneen, and Y.-B. Kim. Towards structured modelling with hyperdag P systems. *International Journal of Computers, Communications and Control*, 2:209–222, 2010.

[27] R. Nicolescu and H. Wu. BFS solution for disjoint paths in P systems. In C. Calude, J. Kari, I. Petre, and G. Rozenberg, editors, *Unconventional Computation*, volume 6714 of *Lecture Notes in Computer Science*, pages 164–176. Springer Berlin / Heidelberg, 2011.

[28] L. Pan, X. Zeng, and X. Zhang. Time-free spiking neural P systems. *Neural Computation*, 23:1320–1342, May 2011.

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

[30] F. Peña-Cantillana, D. Díaz-Pernil, A. Berciano, and M. A. Gutiérrez-Naranjo. A parallel implementation of the thresholding problem by using tissue-like P systems. In P. Real, D. Díaz-Pernil, H. Molina-Abril, A. Berciano, and W. G. Kropatsch, editors, *CAIP (2)*, volume 6855 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 2011.

[31] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Inc., New York, NY, USA, 2010.

[32] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2000.

[33] Z. Yuan and Z. Zhang. Asynchronous spiking neural P system with promoters. In *Proceedings of the 7th international conference on Advanced parallel processing technologies*, APPT'07, pages 693–702, Berlin, Heidelberg, 2007. Springer-Verlag.