ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **17**, Number 1, 2014, 46–62

Complex Objects for Complex Applications

Radu NICOLESCU, Huiling WU

Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand

E-mail: r.nicolescu@auckland.ac.nz,
hwu065@aucklanduni.ac.nz

Abstract. We further develop our earlier Prolog-like complex objects proposal and present several new and improved solutions, for a variety of problems: a faster SAT solution with a fixed number of rules and a set of transformations and components that can build a termination detection algorithm. These results enforce our conjecture that our complex object proposals enable an efficient high-level programming style for P systems.

Key-words: P systems, Prolog terms, complex objects, generic rules, data structures, parallel composition, NP-complete, termination detection, Dijkstra-Scholten.

1. Introduction

A P system is a formal parallel and distributed computational model inspired by the structure and interactions of living cells, introduced by Păun [19]; for a recent overview of the domain, see the comprehensive membrane computing handbook [20]. Essentially, a P system is specified by its membrane structure, symbols and rules. The underlying structure is a *network* such as a digraph, a directed acyclic graph (dag) or a tree (which seems the most studied case). Each node, here better known as *cell*, transforms its content symbols and sends messages to its neighbours using formal rules inspired by *rewriting systems*. Rules of the same cell can be applied in *parallel* (where possible) and all cells work in parallel, either *synchronously* or *asynchronously* (in the sense commonly used in *distributed algorithms* [11, 21] and in Nicolescu et al. [3, 14]).

Here, we make a distinction between (i) generated objects that can be thought, as traditionally in P systems, as being messaged back to the current cell, at the end

of the current step, via a sort of *loopback* channel, and (ii) generated objects which become *immediately* available for the *succeeding* rules, as used by ElGindy et al. [6].

In P systems, the practically very important *modularity* can be achieved by two distinct complementary ways: (i) an *external* modularity, for recursively aggregating groups of cells into higher order P modules, as described by Dinneen et al. [4] — an approach which is not further discussed here, and (ii) an *internal* modularity, possible inside each cell, where we recursively aggregate objects and rules to form higher-order components, as discussed by Nicolescu et al. [16]

We have previously used complex objects to successfully model problems in a wide variety of domains: computer vision [9, 10, 15]; graph theory [18, 6]; distributed algorithms [3, 4, 5, 17, 22]; high-level P systems programming [16]; numerical P systems [16]; NP-complete problems [16].

Traditionally, cells' contents are described as (unstructured) multisets (bags) of *atomic objects* (symbols). Our proposal [16] keeps multisets, but additionally mixes in sub-cellular structures, described as Prolog-like terms, here called *complex objects*. A slightly revised formal definition of complex objects is presented in Section 2. The following sections continue our investigation, presenting new or improved solutions for a variety of more advanced problems: a faster SAT solution with a fixed number of rules, and a set of transformations and components that can build a termination detection algorithm. Without reviewing these here, we note that all our previous applications [16] can be easily redefined to use our revised complex objects definitions; these include fundamental data structures (stacks, queues, trees, dictionaries), high-level control flow (branching, parallel composition, functions), numerical P systems. The ideas of integer arithmetic, compositional properties and high-level programming recall similar ideas also presented (although in different settings) by Alhazov et al. [1] and Manca et al. [12].

Because of space constraints, for the rest of the paper we assume that the reader is already familiar with basic definitions used in tissue-like transition P systems, including state based rules, weak priority, promoters and inhibitors.

2. P systems with complex objects

2.1. Complex objects

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

Technically, our *complex objects*, are Prolog-like first-order *terms*, recursively built from *multisets* of atoms and variables. *Atoms* are typically denoted by lower case letters, such as a, b, c. *Variables* are typically denoted by uppercase letters, such as X, Y, Z.

Terms are either (i) simple atoms, or (ii) atoms (here called *functors*), followed by parenthesized lists of comma-separated "arguments", which are *multisets/bags* of other objects (terms or variables). Terms that do *not* contain variables are called *ground*, e.g.:

- Ground terms: a, $a(b)$, $a(\lambda)$, $a(b, c)$, $a(b, \lambda)$, $a(\lambda, c)$, $a(\lambda, \lambda)$, $a(b(c))$, $a(bc)$, $a(bc(\lambda))$, $a(b(c)d(e))$, $a(b(c), d(e))$, $a(b(c), d(e(\lambda)))$, $a(bc^2, d, eg)$.
- Terms which are not ground: $a(b, X)$, $a(b(X))$, $a(Xc)$, $a(bY)$, $a(XY)$, $a(XdY)$, $a(Xc())$, $a(b(X)d(e))$, $a(b(c), d(Y))$, $a(b(X), d(e(Y)))$.

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

Complex objects can be formally defined by the following grammar:

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

Natural numbers. Natural numbers can be represented via *bags* containing repeated occurrences of the *same* atom. For example, considering that l represents the unary digit, then the following complex objects can be used to describe the contents of a virtual integer *variable* a: $a() = a(\lambda)$ — the value of a is 0; $a(l^3)$ — the value of a is 3. As we show in Section **??**, the arithmetic operations presented by Nicolescu et al. [16] can be adapted to our new representation in a straightforward manner.

Indexed symbols. Complex objects can sometimes be represented as *indexed symbols*, where lower-case indices stand for arguments, e.g. $a_k = a(K)$; this is especially convenient when indices represent numbers or cell IDs (which typically are numbers).

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

- Matching $a(X, eY) = a(b(c), def)$ deterministically creates a single set of unifiers: $X, Y = b(c), df$.
- Matching $a(XY) = a(df)$ non-deterministically creates one of the following four sets of unifiers: $X, Y = \lambda, df$; $X, Y = df, \lambda$; $X, Y = d, f$; $X, Y = f, d$.
- However, matching $a(XY, Y) = a(def, e)$ deterministically creates a single set of unifiers: $X, Y = df, e$.

Performance note. If the rules avoid any matching non-determinism, then this proposal should not affect the performance of P simulators running on existing machines. Assuming that bags are already taken care of, e.g. via hash-tables, our proposed unification probably adds an almost linear factor. Let us recall that, in similar contexts (no occurs check needed), Prolog unification algorithms can run in $O(ng(n))$ steps, where g is the inverse Ackermann function. Our conjecture must be proven though, as (although unlikely) the presence of multisets may still affect the performance.

2.2. Generic rules

By default, rules are applied top-down, in the so-called *weak priority* order. *Rules* may contain *any* kind of terms, ground and not-ground; however, in this proposal, *cells* can only contain *ground* terms. Rules are matched against cell contents using the above discussed *pattern matching*, which involves the rule's left-hand side, promoters and inhibitors. Moreover, the matching is *valid* only if, after substituting variables by their values, the rule's right-hand side contains ground terms only (so *no* free variables are injected in the cell or sent to its neighbours), as illustrated by the following sample scenario:

- The cell's *current content* includes the *ground term*:
  $n(l^{10}, n(l^{20}, f(l^{30}), f(l^{40})), f(l^{50}))$
- The following *rewriting rule* is considered:
  $n(X, n(Y, Y_1, Y_2), f(Z)) \ \rightarrow \ v(X)\, n(Y, Y_1, Y_2)\, v(Z)$
- Our pattern matching determines the following *unifiers*:
  $X = l^{10}$, $Y = l^{20}$, $Y_1 = l^{30}$, $Y_2 = l^{40}$, $Z = l^{50}$.
- This is a *valid* matching and, after *substitutions*, the rule's *right-hand* side gives the *new content*:
  $v(l^{10})\ n(l^{20}, f(l^{30}), f(l^{40}))\ v(l^{50})$

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

current-state *objects*...	$\rightarrow_\alpha$	*target-state*	[*immediate-objects*] ...
		(*in-objects*) ...	(*out-objects*)$_\delta$ ...
		$\mid$ *promoters*...	$\neg$ *inhibitors*...

Where:

- All *objects*, *promoters* and *inhibitors* are *bags of terms*, possibly containing *variables* (which are *matched* as previously described).
- *Out-objects* are sent, at the end of the step, to the cell's structural neighbours. These objects are enclosed in round parentheses which further indicate their destinations, above abbreviated as δ; the most usual scenarious include: $(a) \downarrow_i$ indicates that a is sent to child i (unicast), $(a) \uparrow_i$ indicates that a is sent to

parent i (unicast), $(a)\downarrow_\forall$ indicates that a is sent to all children (broadcast), $(a)\uparrow_\forall$ indicates that a is sent to all parents (broadcast), $(a)\updownarrow_\forall$ indicates that a is sent to all neighbours (broadcast).

- Both *immediate-objects* and *in-objects* remain in the current cell, but there is a subtle difference:
  - *in-objects* become available at the end of the current step only, as in traditional P systems (we can imagine that these are sent via an ad-hoc *loopback* channel);
  - *immediate-objects* become immediately available to (i) to the current rule, if its uses the `max` instantiation mode, and (ii) the succeeding rules (in weak priority order).

  Immediate objects can substantially improve the runtime performance, which could be required for two main reasons: (i) to achieve parity with best traditional algorithms, and (ii) to ensure correctness when proper timing is logically critical.
- Symbol $\alpha \in \{\texttt{min.min}, \texttt{min.max}, \texttt{max.min}, \texttt{max.max}\}$, where `min.min` is the usual default mode, indicates a combined instantiation and rewriting mode, as discussed in Nicolescu et al. [14, 6] (discussion further adapted below).

We often abbreviate by omitting the round parentheses that enclose in-objects. In other words, the traditional in-objects are the default. This respects the tradition and is especially useful for rulesets which do not use immediate objects.

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

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

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

   $$(\rho_1') \quad S_1\ c(c(a)) \rightarrow_{\texttt{min}} S_2\ c(a) \quad \text{or}$$
   $$(\rho_1'') \quad S_1\ c(c(c(a))) \rightarrow_{\texttt{min}} S_2\ c(c(a)).$$

   Using (ρ_1'), cell σ ends with counters $c(a)$, $c(c(a))$, $c(c(c(a)))$. Using (ρ_1''), cell σ ends with counters $c(c(a))$, $c(c(a))$, $c(c(a))$.
2. If $\alpha = \texttt{max.min}$, rule $\rho_{\texttt{max.min}}$ first generates and then applies (in the `min` mode) the following *two* rule instances:

   $$(\rho_2') \quad S_1\ c(c(a)) \rightarrow_{\texttt{min}} S_2\ c(a) \quad \text{and}$$
   $$(\rho_2'') \quad S_1\ c(c(c(a))) \rightarrow_{\texttt{min}} S_2\ c(c(a)).$$

   Using (ρ_2') and (ρ_2''), cell σ ends with counters $c(a)$, $c(c(a))$, $c(c(a))$.

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

$$(\rho_3') \quad S_1\ c(c(a)) \rightarrow_{\texttt{max}} S_2\ c(a) \quad \text{or}$$
$$(\rho_3'') \quad S_1\ c(c(c(a))) \rightarrow_{\texttt{max}} S_2\ c(c(a)).$$

Using (ρ_3'), cell σ ends with counters $c(a)$, $c(a)$, $c(c(c(a)))$. Using (ρ_3''), cell σ ends with counters $c(c(a))$, $c(c(a))$, $c(c(a))$.

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

$$(\rho_4') \quad S_1\ c(c(a)) \rightarrow_{\texttt{max}} S_2\ c(a) \quad \text{and}$$
$$(\rho_4'') \quad S_1\ c(c(c(a))) \rightarrow_{\texttt{max}} S_2\ c(c(a)).$$

Using (ρ_4') and (ρ_4''), cell σ ends with counters $c(a)$, $c(a)$, $c(c(a))$.

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.

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

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

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

3. Arithmetic

Recall our representation for natural numbers. For example, considering that l represents the unary digit, then the following complex objects can indicate that: $a()$ — the value of a is 0; $a(l^3)$ — the value of a is 3.

Fundamental arithmetic operations on natural numbers include:

- $c := a + b$, destructive *addition*:

$$S_1\ a(X)\ b(Y) \quad \rightarrow_{\texttt{min.min}} \quad S_2\ c(XY)$$

- $c := a - b$, destructive *subtraction*:

$$S_1\ a(XY)\ b(Y) \quad \rightarrow_{\mathtt{min.min}} \quad S_2\ c(X)$$

- $c := a * b$, *multiplication*, which destroys a:

$$\begin{array}{lll} S_1 & \rightarrow_{\mathtt{min.min}} & S_2\ c() \\ S_2\ a(lX)\ b(Y)\ c(Z) & \rightarrow_{\mathtt{max.min}} & S_2\ a(X)\ b(Y)\ c(YZ) \\ S_2\ a() & \rightarrow_{\mathtt{min.min}} & S_3 \end{array}$$

- $c,\ d := a\,/\,b,\ a\,\%\,b$, *division*, which destroys a:

$$\begin{array}{lll} S_1 & \rightarrow_{\mathtt{min.min}} & S_2\ c() \\ S_2\ a(XY)\ b(Y)\ c(Z) & \rightarrow_{\mathtt{max.min}} & S_2\ a(X)\ b(Y)\ c(l\,Z) \\ S_2\ a(X) & \rightarrow_{\mathtt{min.min}} & S_3\ d(X) \end{array}$$

Complexity. Note that all the above rules use deterministic matchings only. Thus, additions and subtractions can be performed in single P steps, $O(1)$. However, multiplications and divisions may take longer. For multiplication, the number of steps equals the value of a plus one, whereas for division this is the value of the quotient c plus two. Note that, had we used immediate objects instead of the traditional in-objects, all the above operations would have completed in just one single step.

If desired, non destructive operations can be implemented in a straightforward manner. Alternatively, we can define arithmetic operations using counter stacks, but this could be much slower. These ideas can be extended to define more complete arithmetic packages, e.g. for integer or rational numbers.

4. NP-complete problems

With complex objects, we can solve NP-complete problems using one *single* cell, a *fixed-sized* alphabet and a *fixed-sized* set of generic rules (typically small). In contrast, all proposed solutions which are based on traditional P system models require an exponential runtime number of cells and a polynomial number of symbols and rules (in the size of the input). In fact, a traditional solution describes a *template* for an *infinite* polynomially uniform family of related P systems.

Consider, for example, the SAT problem; see Nagy [13] for a comprehensive overview of this problem and current state-of-art P solutions. Most traditional solutions run in linear time, proportional to the length of the formula. One notable exception is the polynomially uniform family proposed by Gazdag [7], which runs in linear time, proportional to the number of variables, if the formula is given via a suitable encoding. However, this is still a traditional solution, requiring variable numbers of symbols and rules, different for each input formula size.

Our previous solution [16], based on our earlier complex object definition, runs in time linear to the input formula size, uses one single cell, and needs a fixed small number of atomic symbols and rules.

Using our revised complex objects definition (presented here), and encouraged by Gazdac's result [7], we improve our previous solution. We propose a simplified and faster solution, which needs a fixed alphabet and a fixed ruleset and runs in linear time, proportional to the number of variables. We still require a suitable encoding for the input formula; if this is not given, the required encoding can be generated in linear time, proportional to the length of the formula.

We start with an example. Consider the following formula, with $n = 3$ boolean variables:

$$f = (x_1 \vee \bar{x_2}) \wedge (x_1 \vee \bar{x_3}).$$

To map our formula to a fixed vocabulary, we first represent x_i by the term $x(l^i)$ and we label the disjunctions by distinct labels, e.g. α for the first conjunction and β for the second.

Our arguments do not require any particular representation of these labels; we only need that distinct disjunctions are assigned distinct labels (duplicate disjunctions may even receive the same label). For example, such labels can be distinct numbers (which can be generated in linear time), but (as suggested below) the disjunction itself can also serve as a label (possibly using additional functors); this initial labelling phase is not further detailed here.

Then, our sample formula, f, can be expressed by the following cell contents:

$$f = \phi(\surd, x_1, \alpha)\ \phi(\neg, x_2, \alpha)\ \phi(\surd, x_1, \beta)\ \phi(\neg, x_3, \beta), \text{where :}$$

- functor ϕ designates an elementary component of our formula, f;
- functor $\surd$ indicates identity;
- functor $\neg$ indicates negation;
- x_i's are convenience shorthands for terms $x(l^i)$;
- α is a label indicating the first disjunction, e.g. $\alpha = \gamma(l)$ or $\alpha = \vee(\surd(x_1)\,\neg(x_2))$;
- β is a label indicating the second disjunction, e.g. $\beta = \gamma(ll)$ or $\beta = \vee(\surd(x_1)\,\neg(x_3))$.

Intuitively, the ϕ terms can be viewed as rows in a virtual table, ϕ, with three columns, one for each argument, as shown in Table (1a).

To check the satisfiability of our formula, f, we use a simple naive brute force approach and we create $2^n = 2^3 = 8$ mappings (dictionaries), corresponding to all possible truth (0/1) assignments of our $n = 3$ variables. Each mapping is represented by a distinct copy of a complex term, μ, and further referred by a convenience shorthand describing its truth assignments:

$$\begin{aligned}
\mu_{000} &= \mu(m(x(l),0)\ m(x(ll),0)\ m(x(lll),0)) \\
\mu_{001} &= \mu(m(x(l),0)\ m(x(ll),0)\ m(x(lll),1)) \\
\mu_{010} &= \mu(m(x(l),0)\ m(x(ll),1)\ m(x(lll),0)) \\
\mu_{011} &= \mu(m(x(l),0)\ m(x(ll),1)\ m(x(lll),1)) \\
\mu_{100} &= \mu(m(x(l),1)\ m(x(ll),0)\ m(x(lll),0)) \\
\mu_{101} &= \mu(m(x(l),1)\ m(x(ll),0)\ m(x(lll),1)) \\
\mu_{110} &= \mu(m(x(l),1)\ m(x(ll),1)\ m(x(lll),0)) \\
\mu_{111} &= \mu(m(x(l),1)\ m(x(ll),1)\ m(x(lll),1))
\end{aligned}$$

All these 2^n mappings can be built by the following rules, in $n+1$ parallel steps, starting from a single empty mapping, $\mu()$, and a variable $n(N)$, which indicates the number of boolean variables; if this number is not given, it can be easily computed by scanning the given formula (this step is not detailed here):

$$\begin{array}{lll} S_1\, n(lN)\, \mu(M) & \rightarrow_{\texttt{max.min}} & S_1\, n(N)\, \mu(m(x(lN),0)\, M)\, \mu(m(x(lN),1)\, M) \\ S_1\, n() & \rightarrow_{\texttt{max.min}} & S_2 \end{array}$$

For added convenience, we assume the existence of three sets of complex terms, with functors $\neg(,)$, $\vee(,,)$ and $\wedge(,,)$, which can be thought of as builtin read-only internal "tables" for their corresponding boolean operations:

$$\begin{array}{l} \neg(0,1)\ \neg(1,0) \\ \vee(0,0,0)\ \vee(0,1,1)\ \vee(1,0,1)\ \vee(1,1,1) \\ \wedge(0,0,1)\ \wedge(0,1,0)\ \wedge(1,0,1)\ \wedge(1,1,1) \end{array}$$

Next, for each ϕ term in our initial representation of f, we create 2^n new ψ terms, in 1 parallel step, by appending an additional argument, representing one of our 2^n mappings:

$$S_2\ \phi(S,X,D) \quad \rightarrow_{\texttt{max.min}} \quad S_3\ \psi(S,X,D,\mu(M)) \mid \mu(M)$$

Intuitively, the new ψ terms can be viewed as rows in a new virtual table, ψ, with four columns, one for each argument; Tables (1b)&(1c) show $4*2=8$ rows of this new "table", which totals $4*8=32$ rows.

Table 1. Table ϕ and fragments of table ψ

(a) ϕ

$\surd$	x_1	α
$\neg$	x_2	α
$\surd$	x_1	β
$\neg$	x_3	β

(b) $\psi(1:4)$

$\surd$	x_1	α	μ_{000}
$\neg$	x_2	α	μ_{000}
$\surd$	x_1	β	μ_{000}
$\neg$	x_3	β	μ_{000}

(c) $\psi(5:8)$

$\surd$	x_1	α	μ_{001}
$\neg$	x_2	α	μ_{001}
$\surd$	x_1	β	μ_{001}
$\neg$	x_3	β	μ_{001}

Next, we evaluate each of the ψ terms, in 1 parallel step, into a corresponding σ term, which holds its logical value, its corresponding disjunction label and mapping (needed later):

$$\begin{array}{lll} S_3\, \psi(\surd, x_i, D, \mu(m(x_i,V)\, M) & \rightarrow_{\texttt{max.min}} & S_4\, \sigma(V, D, \mu(m(x_i,V)\, M) \\ S_3\, \psi(\neg, x_i, D, \mu(m(x_i,V)\, M) & \rightarrow_{\texttt{max.min}} & S_4\, \sigma(W, D, \mu(m(x_i,V)\, M) \mid \neg(V,W) \end{array}$$

Intuitively, the new σ terms can be viewed as rows in a new virtual table, σ, with three columns, one for each argument; Tables (2b)&(2c) show $4*2=8$ rows of this new "table", which totals $4*8=32$ rows.

Next, we aggregate the values hold by the σ terms, separately for each disjunction and mapping, in $log(2^n)=n$ parallel steps:

$$\begin{array}{lll} S_4\, \sigma(V_1,D,M)\, \sigma(V_2,D,M) & \rightarrow_{\texttt{max.min}} & S_4\, \sigma(W,D,M) \mid \vee(V_1,V_2,W) \\ S_4\, \sigma(V,D,M) & \rightarrow_{\texttt{max.min}} & S_5\, \tau(V,D,M) \end{array}$$

Intuitively, the new τ terms can be viewed as rows in a new virtual table, τ, and represent the values for each formula's disjunction and mapping. Table (2a) shows 4 rows of this new "table".

Table 2. Fragments of tables τ and σ

(a) $\tau(1:4)$

1	α	μ_{000}
1	β	μ_{000}
1	α	μ_{001}
0	β	μ_{001}

(b) $\sigma(1:4)$

$\surd$	0	α	μ_{000}
$\neg$	1	α	μ_{000}
$\surd$	0	β	μ_{000}
$\neg$	1	β	μ_{000}

(c) $\sigma(5:8)$

$\surd$	0	α	μ_{001}
$\neg$	1	α	μ_{001}
$\surd$	0	β	μ_{001}
$\neg$	0	β	μ_{001}

Next, we aggregate the values hold by the τ terms, separately for each mapping, in $log 2^n = n$ parallel steps:

$$\begin{array}{lll} S_5\, \tau(V_1, D_1, M)\, \tau(V_2, D_2, M) & \rightarrow_{\texttt{max.min}} & S_5\, \tau(W, \lambda, M) \mid \wedge(V_1, V_2, W) \\ S_5\, \tau(V, D, M) & \rightarrow_{\texttt{max.min}} & S_6\, \omega(V, M) \end{array}$$

The new ω terms hold the final values of our formula, f, separately for each mapping. Tables (3a)&(3b) show this new "table".

Table 3. Table ω

(a) $\omega(1:4)$

1	μ_{000}
0	μ_{001}
0	μ_{010}
0	μ_{011}

(b) $\omega(5:8)$

1	μ_{100}
1	μ_{101}
1	μ_{110}
1	μ_{111}

Finally, we can return the answer, where S_7 signals *success* and S_8 *failure*:

$$\begin{array}{lll} S_6 & \rightarrow_{\texttt{max.min}} & S_7 \mid \omega(1, M) \\ S_7 & \rightarrow_{\texttt{max.min}} & S_8 \neg\, \omega(1, M) \end{array}$$

In our case, the formula f is *satisfiable*, e.g. for $x_1 = 0$, $x_2 = 0$, $x_3 = 0$ and all cases where $x_1 = 1$. We have used a *single* cell, a *fixed* alphabet, {0, 1, x, l, m, κ, $\vee$, $\wedge$, $\neg$, $\surd$, ϕ, ψ, σ, τ, ω, }, and essentially 11 generic rules and 8 states. In this example, we used only the most basic brute force approach; however, better variants are possible.

A similar approach seems to work well for other NP-complete problems, for example, the graph colouring problem; see Gheorghe et al. [8] for state-of-art P solutions of this problem. We hold the following conjecture:

Conjecture 1. Any NP-complete problems can be solved by a single cell P module with a fixed sized atomic alphabet and a fixed sized set of generic rules.

5. Parallel composition with interaction

We previously introduced parallel composition in [16]. Here we discuss parallel composition with *interaction* of two P systems, Π_1 and Π_2, which can be considered as running in parallel Π_1 and Π_2, where Π_1 "feeds" symbols to Π_2. This parallel composition is essential for cleanly adding a separate control layer, Π_2, over any arbitrary algorithm, Π_1. We use this composition in Section 6, to model a termination detection algorithm.

Consider two P systems, Π_1 and Π_2, which satisfy the following conditions:

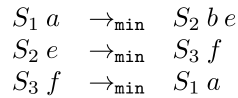
- Π_1 and Π_2 share the same underlying structure (cell network);
- Π_1 and Π_2 use disjoint sets of states (if not, without loss of generality, we can relabel the states to satisfy this condition);
- Π_1 and Π_2 share a set of symbols with three conditions: (i) initially, no left-side symbols or promoters of Π_2 are available; (ii) no rule of Π_2 has empty left-side symbols; (iii) no rule of Π_2 generates symbols of Π_1 or symbols generated by Π_2 do not affect Π_1.

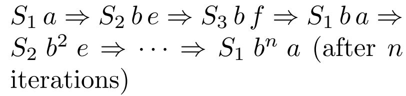
The parallel composition of Π_1 and Π_2 with *interaction*, denoted as $\Pi_1 \triangleright \Pi_2$, is constructed in the following way:

- $\Pi_1 \triangleright \Pi_2$ shares the same structure with Π_1 and Π_2;
- the rules of Π_1 and Π_2 are concatenated, with rules of Π_1 having higher priority than rules of Π_2;
- each state S_i of Π_1 is replaced by complex state $\Theta(S_i, Y)$ and each state S'_i of Π_2 is replaced by complex state $\Theta(X, S'_i)$.

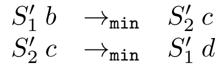
Example 2. Consider two single cell P systems, Π_1 and Π_2. Π_1 cycles over 3 states and each iteration generates one b. Π_2 cycles over 2 states and each iteration transforms one b into one d. Their composition, $\Pi_1 \triangleright \Pi_2$, generates one d in each iteration, where the Π_1 generates one b and the Π_2 component transforms this b into one d.

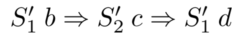
- Π_1, has 3 states and 3 rules:



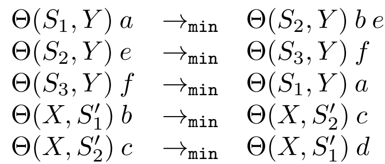


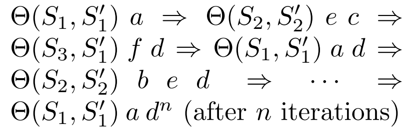
- Π_2, has 2 states and 2 rules:





- $\Pi_1 \triangleright \Pi_2$, has 4 $(= 2 + 2)$ states and 5 $(= 3 + 2)$ rules:





At runtime, $\Pi_1 \triangleright \Pi_2$ seems to use 6 (= 3∗2) states: $\Theta(S_1, S'_1)$, $\Theta(S_2, S'_1)$, $\Theta(S_3, S'_1)$, $\Theta(S_1, S'_2)$, $\Theta(S_2, S'_2)$ and $\Theta(S_3, S'_2)$. However, some states are not reachable and only 3 states are used: $\Theta(S_1, S'_1)$, $\Theta(S_2, S'_2)$ and $\Theta(S_3, S'_1)$.

6. Termination detection

Intuitively, here we discuss an interesting philosophical-like problem, with a metaphorical touch. There are several well defined ways to define the end of a P systems evolution (algorithm). This can be clearly detected from outside, by an external powerful observer, who can continuously probe all cells and all communication channels (without interfering with their current processes). However, can the cells themselves detect this termination?

To ensure this, cells should run a *termination detection* algorithm [11, 21]. This algorithm runs as a separate *control algorithm*, in parallel with the *basic algorithm*, but *interacting* with this at specific points; messages in the control algorithm are called *control messages* and messages in the basic algorithm are called *basic messages*. To make a clean separation, we describe this combination as a parallel composition with interaction (as in Section 5).

At any time during the evolution of a P system, a cell is either *active*, if it has at least one applicable rule; or *passive*, if it cannot apply any (more) rule. A cell is a *source* cell, if it is active when the P system starts to evolve.

In agreement with the classical distributed algorithms theory [11, 21], we make the following assumptions: (i) a message can be sent only by an active cell; (ii) a passive cell can only become active when a message is received; (iii) an active cell can only become passive after applying a rule; (iv) a P system *terminates* when all cells are passive (i.e. no rule is applicable) and all channels are empty (very important in the asynchronous case).

In this section, we illustrate the termination detection by enhancing a simple synchronous Breadth-First-Search (BFS) P algorithm, here called `SyncBFS`, with a control layer based on the well-known Dijkstra-Scholten algorithm [11, 21], here abbreviated `DS`. We first create `SyncBFS`*, a modified version of `SyncBFS`, which creates a few control symbols used by `DS`. Finally, the P solution integrating `SyncBFS` with `DS`, is constructed by a parallel composition with interaction, `SyncBFS`$^* \triangleright$ `DS`. Note that `SyncBFS`, `SyncBFS`* and `DS` are *diffuse* algorithms, which start from the *same single* source cell, σ_s.

6.1. The Dijkstra-Scholten algorithm

The `DS` control algorithm [11, 21] detects termination of the basic algorithm by *dynamically* maintaining a *computation tree*, here called *ds-tree* (which requires an underlying network with duplex channels). `DS` uses the following *detection rules*, organized in two categories: (i) several *extensions of the basic algorithm*: rules that manage *ds-parent* pointers, *ds-counter* integers and *ds-acknowledgment* messages, and (ii) the actual *control layer* (independent of the basic algorithm).

Input: A basic algorithm with one single source cell.
Output: The source cell knows the termination of the basic algorithm.

Extensions of the basic algorithm (pseudocode):

I. Initially, each cell initialises its ds-counter to 0 and the source cell, σ_s, sets its ds-parent as itself.
II. When a cell sends a basic message, it increments its ds-counter by one.
III. When a cell, σ_i, receives a basic message from σ_j: if σ_i has no ds-parent, then it records its ds-parent as σ_j; otherwise, if σ_i has a ds-parent (say σ_k), then it sends a ds-acknowledgment to σ_j (j and k need not be equal).

The above extensions must *not* affect the timing behaviour of the basic algorithm; any change may potentially (but not necessarily) adversely affect the outcome. Ideally, one should also be able to apply this transformation mechanically, without requiring insights into the internals of a specific basic algorithm. The extent of the class of P algorithms where this transformation works is still an open problem.

Control layer of the DS algorithm (pseudocode):

IV. When a cell receives a ds-acknowledgment, it decrements its ds-counter by one and deletes the ds-acknowledgment.
V. When a cell, σ_i, becomes *passive* and its ds-counter becomes 0 (i.e. all its outgoing basic messages have been acknowledged): if $\sigma_i = \sigma_s$, then σ_s knows the algorithm termination; otherwise, if $\sigma_i \neq \sigma_s$, then σ_i sends a ds-acknowledgment to its ds-parent and deletes its ds-parent pointer.

Note that a cell that deletes its ds-parent pointer can again recreate one, if it is reactivated by another basic message. Thus, the ds-tree can repeatedly grow and shrink (in different ways), as a subtree of the underlying network digraph. Rule (V) can be only applied when a cell is *passive* in the basic algorithm. In P systems, this is achieved by a parallel composition with interaction, where the `DS` rules are arranged at a lower priority than the rules of the basic algorithm rules. We now present a P system version of the control layer of the synchronous `DS` algorithm.

P-Algorithm 1 (DS control layer).

Symbols: Cell σ_i uses the following control symbols: p'_j indicates that its ds-parent is σ_j; a is a ds-acknowledgment; w is used for the ds-counters; g signals the algorithm termination. Additionally, σ_i control rules have read-only access to symbol z, conventionally used by the basic algorithm to mark the source, σ_s.

Input: A basic algorithm, which must be enhanced as described by the above rules I-III: (i) the source cell, σ_s, generates one p'_s pointer when it starts computation; (ii) for each sent message, a cell increments its ds-counter by generating one w; (iii) a cell that receives a message from σ_j records its ds-parent as p'_j, if it does not yet contain a ds-parent pointer, or sends an a to σ_j, if it already contains a p'_k.

Output: When the basic algorithm terminates, the source cell, σ_s, contains one g symbol, which signals the algorithm termination.

Rules (only one state is needed, S'_1):

1. $S'_1\ a\ w\ \rightarrow_{\texttt{min.max}}\ S'_1$
2. $S'_1\ p'_j\ \rightarrow_{\texttt{min.min}}\ S'_1\ g\ \neg\ w \mid z$
3. $S'_1\ p'_j\ \rightarrow_{\texttt{min.min}}\ S'_1\ (a) \updownarrow_j\ \neg\ w\ \neg z$

These P rules correspond to rules IV–V of the DS control layer:

IV. Rule 1 considers cell σ_i after receiving a ds-acknowledgment, a: σ_i removes a and decrements its ds-counter by deleting one w.

V. Rules 2–3 consider cell σ_i, after receiving all its due ds-acknowledgments, which is indicated by $\neg\ w$: (rule 2) if $\sigma_i = \sigma_s$, then σ_s generates one g, signaling the algorithm termination; otherwise, (rule 3) if $\sigma_i \neq \sigma_s$, then σ_i sends a ds-acknowledgment, a, to its ds-parent, p'_j, and deletes p'_j.

6.2. Algorithm SyncBFS+DS

`SyncBFS` produces a BFS spanning tree in the synchronous mode. Initially, the source cell broadcasts a visit token. On receiving the visit token, an unvisited cell marks itself as visited, chooses one of the token sending cells as its spanning tree parent (st-parent) and sends its visit token to all other neighbours (i.e. excluding the st-parent)[3]. The algorithm terminates when no more visit tokens are sent, but, without additional help, no cell knows the termination!

As mentioned in Section 6.1, to enable its connection to the `DS` control layer, `SyncBFS` must first be extended, to `SyncBFS`*, with ds-parents, ds-counters and ds-acknowledgments. We present a gradual progression from the basic algorithm `SyncBFS`, to its enhanced version `SyncBFS`* and finally to the composed algorithm (`SyncBFS`* $\triangleright$ `DS`).

P-Algorithms 2 (`SyncBFS`, `SyncBFS`*, (`SyncBFS`* $\triangleright$ `DS`))**.**

Input: Each cell, σ_i, starts with an immutable cell ID symbol, ι_i, and neighbour pointers, n_j's. The source cell, σ_s, is additionally marked with one symbol, z. All cells start with the same set of rules, in the same initial state: S_1 for `SyncBFS` and `SyncBFS`*; $\Theta(S_1, S'_1)$ for (`SyncBFS`* $\triangleright$ `DS`).

Output: All neighbour pointers and cell IDs are intact; σ_s is still marked with one z. Additionally, each cell contains a visited mark, v, and a st-parent pointer, p_j, except σ_s, which contains one p_s, marking it as the root of the spanning tree. All cells end in the same starting state. For (`SyncBFS`* $\triangleright$ `DS`), σ_s contains one g, signaling the algorithm termination.

Other symbols: f marks a visit token holding cell; f_j is a visit token message, forwarded by σ_j; v marks a visited cell. Additionally, `SyncBFS`* and (`SyncBFS`* $\triangleright$ `DS`) use control symbols of `DS` (see P-Algorithm 1).

Rules for `SyncBFS`:

1. $S_1\ \rightarrow_{\texttt{min.min}}\ S_1\ [f_i] \mid \iota_i\ z\ \neg\ v$
2. $S_1\ f_j\ \rightarrow_{\texttt{min.min}}\ S_1\ [f\ v\ p_j]\ \neg\ v$
3. $S_1\ \rightarrow_{\texttt{max.min}}\ S_1\ [w]\ (f_i) \updownarrow_k\ \mid \iota_i\ f\ n_k\ \neg\ p_k$

4. $S_1\ f\ \rightarrow_{\texttt{min.min}}\ S_1$
5. $S_1\ f_j\ \rightarrow_{\texttt{max.min}}\ S_1 \mid v$

Rules for `SyncBFS`* (added control symbols in bold):

1. $S_1\ \rightarrow_{\texttt{min.min}}\ S_1\ [f_i] \mid \iota_i\ z \neg v$
2. $S_1\ f_j\ \rightarrow_{\texttt{min.min}}\ S_1\ [f\ v\ p_j\ \boldsymbol{p'_j}] \neg v$
3. $S_1\ \rightarrow_{\texttt{max.min}}\ S_1\ [w]\ (f_i) \updownarrow_k\ \mid \iota_i\ f\ n_k \neg p_k$
4. $S_1\ f\ \rightarrow_{\texttt{min.min}}\ S_1$
5. $S_1\ f_j\ \rightarrow_{\texttt{max.min}}\ S_1\ \boldsymbol{(a) \updownarrow_j}\ \mid v$

Rules for (`SyncBFS`$^* \triangleright$ `DS`): The ruleset consists of the concatenation of:

- First, the rules of `SyncBFS`*, transformed by replacing S_1 with $\Theta(S_1, Y)$;
- Followed by the rules of `DS`, transformed by replacing S'_1 with $\Theta(X, S'_1)$ (transformation not detailed here).

Brief description of `SyncBFS` and (`SyncBFS`$^* \triangleright$ `DS`) rules 1–5: Proposition 3 shows that `SyncBFS`* rules 1–5 cover rules I–III of the `DS` extensions of the basic algorithm `SyncBFS`:

I. Rule 1: the source cell, σ_s, generates one f_s, which is next used to set its ds-parent, p_s, in the same step (rule 2).
II. Rule 3: When cell σ_i holds the f token, it sends an f_i to each neighbour, except its st-parent, and generates one w for each sent f_i.
III. Rules 2 and 5 consider a cell, σ_i, after receiving f_j from σ_j:
   (a) Rule 2: σ_i generates p'_j, if no v exists $\equiv$ if no p'_k exists.
   (b) Rule 5: σ_i sends a to σ_j, if v exists $\equiv$ if a p'_k exists.

Proposition 3. In any run of (`SyncBFS`$^* \triangleright$ `DS`), the following statements hold:

I. Cell σ_i contains a v marker iff σ_i contains a st-parent pointer, p_k.
II. If cell σ_i contains an f_j symbol, then (σ_i contains an v marker iff σ_i contains a ds-parent pointer, p'_k).

Why use immediate objects? Symbols f_i, f, v, p_j, p'_k and w are generated by the first rules and then *immediately* used by the following rules, in the *same step.* Generating these as immediate objects ensures that the control layer rules do not delay or otherwise perturb the evolution of the basic algorithm.

Figure 1 shows a simple (`SyncBFS`$^* \triangleright$ `DS`) example. (a) The source cell, σ_1, broadcasts its visit token and sets its ds-counter to 3. (b) On receiving σ_1's visit token, each of the unvisited cells, σ_2, σ_3 and σ_4, marks itself as visited, sets its st-parent and ds-parent as σ_1, sends its visit tokens to all other neighbours and sets its ds-counter to 2. (c) On receiving these visit tokens, cells σ_2, σ_3 and σ_4, which are already visited and have ds-parents, ignore these tokens and send back ds-acknowledgments to the token senders. (d) Cells σ_2, σ_3 and σ_4 receive all their due ds-acknowledgments,

i.e. their ds-counters reach 0, so each of them sends a ds-acknowledgment to its ds-parent, σ_1. (e) The source cell, σ_1, receives all its due ds-acknowledgments and thus knows the algorithm termination. Table 4 shows the initial and final configurations of ($\texttt{SyncBFS}^* \triangleright \texttt{DS}$), for Figure 1.

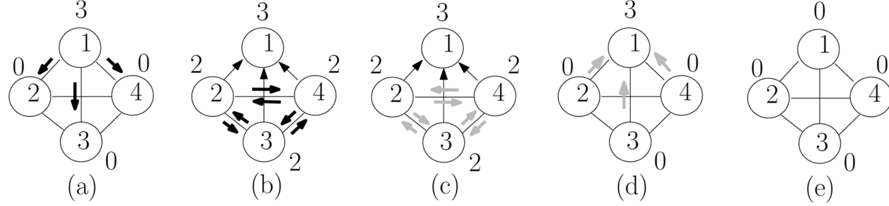


Fig. 1. ($\texttt{SyncBFS}^* \triangleright \texttt{DS}$) example. Edges with arrows: ds-tree arcs; black arrows near edges: basic messages (visit tokens); gray arrows near edges: ds-acknowledgments; the number beside each cell: its ds-counter.

Table 4. Initial and final configurations of ($\texttt{SyncBFS}^* \triangleright \texttt{DS}$) for Fig. 1.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_1, S_1')\ z\ \iota_1$ $n_2\ n_3\ n_4$	$\Theta(S_1, S_1')\ \iota_2$ $n_1\ n_3\ n_4$	$\Theta(S_1, S_1')\ \iota_3$ $n_1\ n_2\ n_4$	$\Theta(S_1, S_1')\ \iota_4$ $n_1\ n_2\ n_3$
4	$\Theta(S_1, S_1')\ z\ \iota_1\ v$ $p_1\ n_2\ n_3\ n_4\ \boldsymbol{g}$	$\Theta(S_1, S_1')\ \iota_2\ v$ $p_1\ n_1\ n_3\ n_4$	$\Theta(S_1, S_1')\ \iota_3\ v$ $p_1\ n_1\ n_2\ n_4$	$\Theta(S_1, S_1')\ \iota_4\ v$ $p_1\ n_1\ n_2\ n_3$

7. Conclusion

Despite their intrinsic theoretical and modelling power, traditional P systems are a bit brittle for modelling complex algorithms or applications, which seem to require large and varying size unstructured rulesets, that can be difficult to develop and verify. We want to show that this need not be the case, that there are ways to increase their usability.

The gathered evidence suggests that complex objects enable a higher-level programming style, with fixed sized and better structured rulesets (and alphabets) and including features of modern functional, parallel and distributed programming. The formal P rulesets are typically crisper or comparable to the best existing pseudocode. We intend to continue this investigation and make this work more complete, for example, by modelling a termination detection algorithm for the more complex asynchronous case. An interesting open problem is to characterize the class of P algorithms which can be mechanically transformed by rules I-III of Section 6.1.

Although much work remains to be done, we hope that our extensions will be directly mapped to modern or emerging computing platforms (bypassing a possible translation to traditional simpler objects and rules), which will enable more efficient general purpose simulators.

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

References

[1] ALHAZOV A., BONCHIŞ C., CIOBANU G., IZBAŞA C., *Encodings and arithmetic operations in P systems*, in M. Gutierrez-Naranjo, G. Păun, A. Riscos-Nunez, F. J. Romero-Campero, editors, *Brainstorming Week on Membrane Computing*, volume 2, pp. 1–28, Universidad de Sevilla, 2006.

[2] ALHAZOV A., COJOCARU S., GHEORGHE M., ROGOZHIN Y., editors, *14th International Conference on Membrane Computing*, CMC14, Chişinău, Moldova, Au- gust 20-23, 2013, Proceedings. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, Chişinău, 2013.

[3] BĂLĂNESCU T., NICOLESCU R., WU H., *Asynchronous P systems*, International Journal of Natural Computing Research, **2**(2):1–18, 2011.

[4] DINNEEN M. J., KIM Y.-B., NICOLESCU R., *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, pp. 175–197. Springer-Verlag, Berlin Heidelberg, 2010.

[5] DINNEEN M. J., KIM Y.-B., NICOLESCU R., *P systems and the Byzantine agreement*, Journal of Logic and Algebraic Programming, **79**(6):334–349, 2010.

[6] ELGINDY H., NICOLESCU R., WU H., *Fast distributed DFS solutions for edge- disjoint paths in digraphs*, in E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil, editors, *Membrane Computing*, volume **7762** of Lecture Notes in Computer Science, pp. 173–194, Springer Berlin Heidelberg, 2013.

[7] GAZDAG Z., *Solving SAT by P systems with active membranes in linear time in the number of variables*, in Alhazov et al. [2], pp. 167–180.

[8] GHEORGHE M., IPATE F., LEFTICARU R., PEREZ-JIMENEZ M., TURCANU A., VALENCIA CABRERA L., GARCIA-QUISMONDO M., MIERLA L., *3-col problem modelling using simple kernel P systems*, Int. J. Comput. Math., **90**(4):816–830, Apr. 2013.

[9] GIMELFARB G., NICOLESCU R., RAGAVAN S., *P systems in stereo matching*, in P. Real, D. Diaz-Pernil, H. Molina-Abril, A. Berciano, W. Kropatsch, editors, *Computer Analysis of Images and Patterns*, volume **6855** of Lecture Notes in Computer Science, pp. 285–292, Springer Berlin Heidelberg, 2011.

[10] GIMELFARB G., NICOLESCU R., RAGAVAN S., *P system implementation of dynamic programming stereo*, Journal of Mathematical Imaging and Vision, **47**(1–2):13–26, 2013.

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

[12] MANCA V., LOMBARDO R., *Computing with multi-membranes*, in M. Ghe- orghe, G. Paun, G. Rozenberg, A. Salomaa, S. Verlan, editors, *Membrane Computing*, volume **7184** of Lecture Notes in Computer Science, pp. 282–299, Springer Berlin Heidelberg, 2012.

[13] NAGY B., *On efficient algorithms for SAT*, in E. Csuhaj-Varju, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil, editors, *Membrane Computing*, volume **7762** of Lecture Notes in Computer Science, pp. 295–310, Springer Berlin Heidelberg, 2013.

[14] NICOLESCU R., *Parallel and distributed algorithms in P systems*, in M. Gheorghe, G. Paun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing, CMC 2011, Revised Selected Papers*, volume **7184** of Lecture Notes in Computer Science, pp. 35–50. Springer Berlin Heidelberg, 2012.

[15] NICOLESCU R., GIMELFARB G., MORRIS J., GONG R., DELMAS P., *Regularising ill-posed discrete optimisation: Quests with P systems*, Fundam. Inf., **131**(3–4):465–483, 2014.

[16] NICOLESCU R., IPATE F., WU H., *Towards high-level P systems programming using complex objects*, in Alhazov et al. [2], pp. 255–276.

[17] NICOLESCU R., WU H., *BFS solution for disjoint paths in P systems*, in C. Calude, J. Kari, I. Petre, G. Rozenberg, editors, *Unconventional Computation*, volume **6714** of Lecture Notes in Computer Science, pp. 164–176. Springer Berlin Heidelberg, 2011.

[18] NICOLESCU R., WU H., *New solutions for disjoint paths in P systems*, Natural Computing, **11**:637–651, 2012.

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

[20] PĂUN G., ROZENBERG G., SALOMAA A., *The Oxford Handbook of Membrane Computing*, Oxford University Press, Inc., New York, NY, USA, 2010.

[21] TEL G., *Introduction to Distributed Algorithms*, Cambridge University Press, 2000.

[22] WU H., *Minimum spanning tree in P systems*, in L. Pan, G. Păun, T. Song, editors, *Proceedings of the Asian Conference on Membrane Computing (ACMC2012)*, pp. 88–104, Wuhan, China, 2012, Huazhong University of Science and Technology.