Kernel P Systems Modelling, Testing and Verification

Marian Gheorghe[1], Rodica Ceterchi[2], Florentin Ipate[2,3] and Savas Konur[1]

[1] School of Electrical Engineering and Computer Science, University of Bradford
Bradford BD7 1DP, UK
{m.gheorghe, s.konur}@bradford.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, rceterchi@gmail.com
[3] Department of Computer Science, University of Piteşti
Str Targul din Vale, nr 1, 110040, Argeş, Romania

Summary. A kernel P system (kP system, for short) integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, consequently, it provides a framework for analyzing these models. In this paper, we illustrate the modeling capabilities of kernel P systems by showing how other classes of P systems can be represented with this formalism and providing a number of kP system models for sorting algorithms. Furthermore, the problem of testing systems modelled as kP systems is also discussed and a test generation method based on automata is proposed. We also demonstrate how formal verification can be used to validate that the given models work as desired.

1 Introduction

Membrane systems were introduced in [27] as a new natural computing paradigm inspired by the structure and distribution of the compartments of living cells, as well as by the main bio-chemical interactions occurring within compartments and at the inter-cellular level. They were later also called *P systems*. An account of the basic fundamental results can be found in [28] and a comprehensive description of the main research developments in this area is provided in [29]. The key challenges of the membrane systems area and a discussion on some future research directions, are available in a more recent survey paper [20].

In recent years, significant progress has been made in using P systems to model and simulate systems and problems from various areas. However, in order to facilitate the modelling, in many cases various features have been added in an ad-hoc manner to these classes of P systems. This has led to a multitude of P systems variants, without a coherent integrating view. The newly introduced concept of *kernel P systems (kP systems)* [16, 17] provides a response to this problem. A kP system integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, consequently, it provides a framework for analyzing these models. Furthermore, the expressive power of these systems has been illustrated by a number of representative case studies [19, 17]. The kP system model is supported by a modelling language, called kP-Lingua, capable of mapping a kP system specification into a machine readable representation. Furthermore, kP systems are supported by a software framework, kPWorkbench [21], which integrates a set of related simulation and verification tools and techniques.

Another complementary method to simulation and verification is testing, a major activity in the lifecycle of software systems. In practice, software products are almost always validated through testing. Testing has been discussed for cell-like P systems and various strategies, such as rule coverage based and automata based techniques, have been proposed [15, 24]. Until now, however, testing has not been discussed in the context of kP systems.

In this paper we further illustrate the modeling capabilities of kernel P systems by showing that other classes of P systems can be represented with this formalisms and by providing a number of kP system models for sorting algorithms. We present in this paper the relationship between kP systems and active membrane systems with electrical charges, whereas in [16, 17, 18] we have also investigated the relationship with neural-like P systems. We also study here the relationship between kP systems and P systems with symport/antiport rules. Furthermore, the problem of testing systems modelled as kP systems is also discussed and a test generation method based on automata is proposed. We also demonstrate how formal verification can be used to validate that the given models work as desired.

2 kP Systems - Main Concepts and Definitions

We consider that standard P system concepts such as strings, multisets, rewriting rules, and computation are well-known and refer to [28] for their formal notations and precise definitions. The kP system concepts and definitions introduced below are from [16, 17]; some are slightly changed and this will be mentioned.

Definition 1. *T is a* set of compartment types*, $T = \{t_1, \dots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \le i \le s$, consists of a set of rules, R_i, and an execution strategy, σ_i, defined over $Lab(R_i)$, the labels of the rules of R_i.*

Remark 1. The compartments that appear in the definition of the kP systems will be instantiated from these compartment types. The types of rules and the execution strategies will be discussed later.

Definition 2. *A* kernel P (kP) system *of degree n is a tuple*

$$k\Pi = (A, \mu, C_1, \dots, C_n, i_0),$$

where A is a finite set of elements called objects*; μ defines the* initial membrane structure*, which is a graph, (V, E), where V are vertices indicating components, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a* compartment *of the system consisting of a compartment type from T and an* initial multiset*, w_i over A; i_o is the* output compartment *where the result is obtained.*

2.1 kP System Rules

The discussion below assumes that the rules we refer to belong to the same compartment, C_i.

Each rule r may have a **guard** g which refers to the multiset where the rule is applied to. Its generic form is $r\ \{g\}$. The rule r is applicable to a multiset w when its left hand side is contained into w and g is true for w.

The guards are constructed using multisets over A, as operands, and relational and Boolean operators. Let us first introduce some notations.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w. Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, a^n a multiset and $r\ \{g\}$ a rule with guard g. We first introduce an *abstract relational expression* which is evaluated for any multiset where the rule is applied to.

Definition 3. *If g is the* abstract relational expression γa^n *and w is the multiset it refers to, then the guard denotes the* relational expression $|w|_a \gamma n$*. The guard g is true for the multiset w if $|w|_a \gamma n$ is true.*

One can consider the Boolean operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction), listed with respect to the decreasing precedence order. *Abstract Boolean expressions* are obtained by connecting abstract relational expressions by Boolean operators.

Definition 4. *If g is the* abstract Boolean expression *and the current multiset is w, then the guard denotes the* Boolean expression *for w, obtained by replacing abstract relational expressions with relational expressions for w. The guard g is true for the multiset w when the Boolean expression for w is true.*

Definition 5. *A guard is: (i) one of the Boolean constants true or $false$; (ii) an abstract relational expression; or (iii) an abstract Boolean expression.*

Example 1. If g is the guard $\geq a^5 \wedge \geq b^3 \vee \neg > c$ and w a multiset it refers to, then g is true in w if it has at least 5 a's and 3 b's or no more than one c.

Definition 6. *A rule from a compartment $C_{l_i} = (t_{l_i}, w_{l_i})$ can have one of the following types:*

- *(a)* **rewriting and communication** *rule: $x \to y\ \{g\}$,*
  where $x \in A^+$ and y has the form $y = (a_1, t_1) \ldots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j indicates a compartment type from T – see Definition 2 – with instance

compartments linked to the current compartment; t_j might also indicate the type of the current compartment, t_{l_i}, (in this case it is not present on the right hand side of the rule); if a link does not exist (i.e., there is no link between the two compartments in E) then the rule is not applied; if a target, t_j, refers to a compartment type that has more than one instance connected to C_{l_i}, then one of them will be non-deterministically chosen;

- *(b)* **structure changing rules***; the following types of rules are considered:*
  - *(b1)* **membrane division** *rule:* $[x]_{t_{l_i}} \rightarrow [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}}\ \{g\}$,
    where $x \in A^+$ and $y_j \in A^*$; the compartment C_{l_i} will be replaced by p compartments; the j-th compartment, instantiated from the compartment type t_{i_j} contains the same objects as C_{l_i}, but x, which will be replaced by y_j; all the links of C_{l_i} are inherited by each of the newly created compartments;
  - *(b2)* **membrane dissolution** *rule:* $[]_{t_{l_i}} \rightarrow \lambda\ \{g\}$;
    the compartment C_{l_i} will be destroyed together with its links;
  - *(b3)* **link creation** *rule:* $[x]_{t_{l_i}}; []_{t_{l_j}} \rightarrow [y]_{t_{l_i}} - []_{t_{l_j}}\ \{g\}$;
    the current compartment is linked to a compartment of type t_{l_j} and x is transformed into y; if more than one instance of the compartment type t_{l_j} exists then one of them will be non-deterministically picked up; g is a guard that refers to the compartment instantiated from the compartment type t_{l_1};
  - *(b4)* **link destruction** *rule:* $[x]_{t_{l_i}} - []_{t_{l_j}} \rightarrow [y]_{t_{l_i}}; []_{t_{l_j}}\ \{g\}$;
    is the opposite of link creation and means that the compartments are disconnected.

The membrane division is defined slightly differently here compared to [16, 17]. Currently, the right hand side of the rule uses simple multisets with no target compartments, as they were initially introduced in [16, 17].

2.2 kP System Execution Strategies

In kP systems the way in which rules are executed is defined for each compartment type t from T – see Definition 1 and Remark 1. As in Definition 1, $Lab(R)$ is the set of labels of the rules R.

Definition 7. *For a compartment type $t = (R, \sigma)$ from T and $r \in Lab(R)$, $r_1, \dots, r_s \in Lab(R)$, the* execution strategy, *σ, is defined by the following*

- *$\sigma = \lambda$, means no rule from the current compartment will be executed;*
- *$\sigma = \{r\}$ – the rule r is executed;*
- *$\sigma = \{r_1, \dots, r_s\}$ – one of the rules labelled $r_1, \dots, r_s$ will be chosen non-deterministically and executed; if none is applicable then none is executed; this is called* alternative *or* choice*;*
- *$\sigma = \{r_1, \dots, r_s\}^*$ – the rules are applied an arbitrary number of times (* arbitrary parallelism*);*
- *$\sigma = \{r_1, \dots, r_s\}^\top$ – the rules are executed according to* maximal parallelism *strategy x;*

- *$\sigma = \sigma_1 \& \ldots \& \sigma_s$, means executing sequentially $\sigma_1, \ldots, \sigma_s$, where σ_i, $1 \leq i \leq s$, describes any of the above cases, namely λ, one rule, a choice, arbitrary parallelism or maximal parallelism; if one of σ_i fails to be executed then the rest is no longer executed;*
- *for any of the above σ strategy only one single structure changing rule is allowed.*

Arbitrary parallelism and maximal parallelism for rewriting and communication rules, as well as for structure changing rules (cell division, dissolution), are discussed in [29].

Remark 2. In certain cases the operator & will be ignored and the sequential execution will be denoted as $\sigma = \sigma_1 \ldots \sigma_s$.

Remark 3. A computation, as usual in membrane computing, is defined as a sequence of finite steps starting from the initial configuration, with the initial multisets distributed in compartments. In each step the rules are selected according to the execution strategy and this is given by the execution strategy in each compartment. The result of a computation will be the number of objects collected in the output compartment. For a kP systems $k\Pi$, the set of all these numbers will be denoted by $M(k\Pi)$.

Remark 4. When a terminal alphabet, F, is considered, the result of a computation will be the number of objects from F collected in the output compartment and this will be denoted by $M_t(k\Pi)$

3 kP Systems and Other Classes of P Systems

In this section we will investigate the relationship between kP systems and P systems with active membranes, but other relevant classes of P systems will be also considered, especially those with various applications, such as symport/antiport P systems. In [17, 18] neural-like P systems have been also considered.

3.1 P Systems with Active Membranes versus kP Systems

We study how P systems with active membranes are simulated by kP systems. In this case we are dealing with a cell-like system, so the underlying structure is a tree and we have a set of labels (types) for the compartments of the system. The way the relationship between these P systems is presented in the sequel is a natural extension of the method proposed in [16, 17, 18]. In the previous investigations the set of objects from the output compartment has been mixed up with the rest of the objects of the system. In this investigation we separate the objects corresponding to the output compartment and provide a more consistent notation for the kP system involved. We also deal in this investigation with active membrane systems with an upper bound for the number of active components

Definition 8. *A P system with active membranes of initial degree* n *is a tuple (see [29], Chapter 11)* $\Pi = (O, H, \mu, w_{1,0}, \dots, w_{n,0}, R, i_0)$ *where:*

- O *is an alphabet of objects,* $w_{1,0}, \dots, w_{n,0}$ *are the initial strings in the* n *initial compartments and* i_0 *is the output compartment;*
- H *is the set of labels for compartments;*
- μ *defines the tree structure associated with the system;*
- R *consists of rules of the following types:*
  - *(a)* rewriting rules*:* $[u \to v]_h^e$*, for* $h \in H$*,* $e \in \{+, -, 0\}$ *(set of electrical charges),* $u \in O^+$*,* $v \in O^*$*;*
  - *(b)* in communication rules*:* $u[]_h^{e_1} \to [v]_h^{e_2}$*, for* $h \in H$*,* $e_1, e_2 \in \{+, -, 0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(c)* out communication rules*:* $[u]_h^{e_1} \to []_h^{e_2} v$*, for* $h \in H$*,* $e_1, e_2 \in \{+, -, 0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(d)* dissolution rules*:* $[u]_h^e \to v$*, for* $h \in H \backslash \{s\}$*,* s *denotes the skin membrane (the outmost one),* $e \in \{+, -, 0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(e)* division rules for elementary membranes*:* $[u]_h^{e_1} \to [v]_h^{e_2} [w]_h^{e_3}$*, for* $h \in H$*,* $e_1, e_2, e_3 \in \{+, -, 0\}$*,* $u \in O^+$*,* $v, w \in O^*$*;*

The rules are executed in accordance with the maxim parallelism, but in each compartment only one of the rules (b)-(e) is executed. In the sequel we assume that the output compartment is neither dissolved nor divided. The result of a computation, obtained in i_0 is denoted by $M(\Pi)$.

The following result shows how the computation of a P system with active membranes starting with n_1 compartments and an upper bound to the number of active compartments can be performed by a kP system using only rewriting and communication rules. A first idea of this result has been given in [16, 17, 18].

Theorem 1. *If* Π *is a P system with active membranes having* n_1 *initial compartments and an upper bound to the number of active compartments in any computation, then there exists a kP system,* $k\Pi$*, of degree* 2 *and using only rewriting and communication rules, such that* $M(\Pi) = M(k\Pi)$*.*

Proof. Let $\Pi = (O, H, \mu, w_{1,0}, \dots, w_{n_1,0}, R, i_0)$ be a P system with active membranes of initial degree n_1. Initially, the polarizations of the n_1 compartments are all 0, i.e., $e_1 = \ldots = e_{n_1} = 0$.

We will build a kP system with two compartments. Compartment C_1 will capture the contents and rules of all the compartments of Π. The other compartment, C_2 will be associated to i_0 and this will collect the result.

We will need to keep track of a dynamic system of membranes, since we have dissolution and division of elementary membranes. We will identify a membrane by a pair (i, h) where $i \in I$ is an index associated with an instance of the membrane and $h \in H$ is its label. We use the index in addition to the label as the same label might appear several times in the system, especially after a membrane division rule has been applied. We work under the assumption that I is finite. Its cardinal is equal to the maximum number of active membranes that may appear in any computation – this is assumed to have an upper limit. We let $i_0 \in I$ and $i_0 \in H$. We will denote by $(I \times H)_c$ the currently used pairs (i, h). We assume that for any $(i, h) \in (I \times H)_c$ and $(j, h') \in (I \times H)_c$, we have $i \neq j$. This way we make sure that the cardinal of $(I \times H)_c$ is always at most the cardinal of I. Whenever a membrane dissolution takes place, its index and label are removed from $(I \times H)_c$. When a membrane division rule is applied the index and label of the divided compartment are removed from $(I \times H)_c$ and two new values of indices with the same label are selected and added to the set $(I \times H)_c$. The tuple (i_0, i_0) is always in $(I \times H)_c$

We will codify a compartment $[w]_h^e$ by two tuples $< e, i, h >$ and $< w, i, h >$, with $(i, h) \in (I \times H)_c$, and where, for a multiset $w = a_1 \dots a_m$, $< w, i, h >$ denotes $< a_1, i, h > \dots < a_m, i, h >$. These tuples appear in C_1. When $h = i_0$ then in addition to the tuples present in C_1, in C_2, for $[w]_{i_0}^e$ we have e and w. For a compartment with label h and electrical charge e in Π there is only one tuple $< e, i, h >$ in C_1, when $h \neq i_0$, or an e in C_2, otherwise.

By $p(i, h)$ we denote the parent of the membrane with label h and of index i. If $p(i, h) = (i', h')$ it means that the membrane with label h' and index i' is the parent of the membrane with label h and index i. By $< x, p(i, h) >$ and $< e, p(i, h) >$ we denote the tuples $< x, i', h' >$ and $< e, i', h' >$, respectively.

A new symbol, δ, will be used for the membrane dissolution and division to control the transfer of objects after these rules have been applied. Hence, we will use the guard

$$\overline{= \delta_{all}} := \bigwedge (\neg =< \delta, i, h >| \; i \in I, h \in H).$$

We also introduce a guard checking that the symbols γ_1 ans γ_2, related with the communication with the output compartment, i_0, do not appear in the current multiset:

$$\overline{= \gamma_{all}} := (\neg = \gamma_1) \wedge (\neg = \gamma_2).$$

We construct $k\Pi$ using $T = \{t_1, t_2\}$, where $t_j = (R'_j, \sigma_j)$ (where R'_j and σ_j will be defined later), $1 \leq j \leq 2$, as follows: $k\Pi = (A, \mu', C_1, C_2, 2)$, where the elements of the system are given below.

- μ' is the graph with nodes C_1, C_2 and the edge linking them;
- The alphabet is

$$\begin{aligned} A = O \cup \{0, 0', +, +', -, -', \gamma_1, \gamma_2\} \\ \cup (\textstyle\bigcup_{(i,h) \in I \times H} (\{< a, i, h >| \; a \in O \cup \{\delta\}\} \cup \{< e, i, h >| \; e \in \{0, +, -\}\})) \end{aligned}$$

- $C_j = (t_j, w'_{j,0} w''_{j,0})$, $1 \leq j \leq 2$ and C_2 is the output compartment.
  – The initial multiset, $w'_{1,0} w''_{1,0}$, is given by

$$w'_{1,0} =< w_{1,0}, 1, h_1 > \dots < w_{n_1,0}, n_1, h_{n_1} > \overline{w_{i_0,0}}$$

  where $\overline{w_{i_0,0}}$ means that $w_{i_0,0}$ does not appear in the initial multiset of C_1 (it will appear in C_2).

$$w''_{1,0} = \{< e_1, 1, h_1 > \dots < e_{n_1}, n_1, h_{n_1} > \overline{e_{i_0}}$$

where $e_1 = \ldots = e_{n_1} = 0$, for all the initial multisets and initial membranes of Π, and, similar to the above case, $\overline{e_{i_0}}$ means that e_{i_0} does not appear in the initial multiset. The initial multiset $w'_{2,0}w''_{2,0}$, is given by

$$w'_{2,0} = w_{i_0,0}, w''_{2,0} = e_{i_0}.$$

Initially, the indices $(I \times H)_1 = \{(1, h_1) \ldots (n_1, h_{n_1})\} \setminus \{(i_0, i_0)\}$ are used in association with compartment C_1 and (i_0, i_0) for C_2. The currently used indices are $(I \times H)_c = (I \times H)_1 \cup \{(i_0, i_0)\}$.

– R'_1 and R'_2 contain the rules below.

(a.1) For each $(i, h) \in I \times H \setminus \{(i_0, i_0)\}$ and each rule $[u \to v]^e_h \in R$, $e \in \{+, -, 0\}$, we add to R'_1 the rule $< u, i, h > \to < v, i, h > \{= < e, i, h > \wedge \overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$; these rules are applied only when the polarization e appears in the compartment with index i and label h and none of the (δ, j, h'), γ_1, γ_2 appears, i.e., no dissolution or division has started and no communication with the output compartment, i_0, takes place – see below.

(a.2) For $(i, h) = (i_0, i_0)$, we add to R'_1 the rule $< u, i_0, i_0 > \to < v, i_0, i_0 > \{= < e, i_0, i_0 > \wedge \overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$ and the rule $u \to v \{= e \wedge \overline{\equiv \gamma_{all}}\}$ to R'_2.

(b.1) For each $(i, h) \in I \times H \setminus \{(i_0, i_0)\}$, such that $p(i, h) \neq (i_0, i_0)$, and each rule $u[]^{e_1}_h \to [v]^{e_2}_h \in R$, $e_1, e_2 \in \{+, -, 0\}$, we add to R'_1 the rule $< u, p(i, h) >< e_1, i, h > \to < v, i, h >< e_2, i, h > \{\overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$; these rules will transform $< u, p(i, h) >$ corresponding to u from the parent compartment to $< v, i, h >$ corresponding to v from the compartment with index i and label h; the polarization is changed; as there is only one object $< e_1, i, h >$, it follows that only one single rule corresponding to the compartment can be applied at any moment of the computation.

(b.2) When $(i, h) = (i_0, i_0)$, then the rules added to R'_1 are $< u, p(i_0, i_0) >< e_1, i_0, i_0 > \to < v, i_0, i_0 >< e_2, i_0, i_0 > (ve'_2\gamma_1, 2)\gamma_1 \{\overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$ and $\gamma_1 \to \lambda$; and the rules added to R'_2 are $e'_2 \to e_2\{= \gamma_1\}$ and $\gamma_1 e \to \lambda$, $e \in \{0, +, -\}$. The first rule apart from simulating the communication rule, also introduces γ_1 in both compartments. In C_2 it helps changing the polarization of it and in C_1 it helps with the synchronisation of the computation. Then the symbol disappears.

(b.3) When $p(i, h) = (i_0, i_0)$, then we add to R'_1 the rules $< u, i_0, i_0 >< e_1, i, h > \to < v, i, h >< e_2, i, h > (\gamma_2, 2)\gamma_2 \{\overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$ and $\gamma_2 \to \lambda$. The rule $u\gamma_2 \to \lambda$ is added to R'_2. Similar to $(b.2)$, γ_2 is introduced in both compartments and in C_2 it helps removing u.

(c.1) For each $(i, h) \in I \times H \setminus \{(i_0, i_0)\}$, such that $p(i, h) \neq (i_0, i_0)$, and each rule $[u]^{e_1}_h \to []^{e_2}_h v \in R$, $e_1, e_2 \in \{+, -, 0\}$, we add the rule $< u, i, h >< e_1, i, h > \to < v, p(i, h) >< e_2, i, h > \{\overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$.

(c.2) When $(i, h) = (i_0, i_0)$, then we add to R'_1 the rule $< u, i_0, i_0 >< e_1, i_0, i_0 > \to < v, p(i_0, i_0) >< e_2, i_0, i_0 > (e'_2\gamma_1, 2)\gamma_1 \{\overline{= \delta_{all}} \wedge \overline{\equiv \gamma_{all}}\}$. As in $(b.2)$, we use $\gamma_1 \to \lambda$ in R'_1 and $e'_2 \to e_2\{= \gamma_1\}$ in R'_2. We need to add to R'_2 the rule $u\gamma_1 e \to \lambda$. The rules make sure that in C_1 we simulate the communication rule and in C_2 u disappears and the polarization is changed to e_2.

(c.3) When $p(i,h) = (i_0, i_0)$, then the rule added to R'_1 is $< u,i,h >< e_1,i,h > \to < v,i_0,i_0 >< e_2,i,h > (v,2)$ $\{\overline{= \delta_{all}} \wedge \overline{= \gamma_{all}}\}$. This rule simulates the communication rule and introduces v into C_2.

(d.1) for each $(i,h) \in I \times H \setminus \{(i_0,i_0)\}$, such that $p(i,h) \neq (i_0,i_0)$, and each rule $[u]_h^e \to v \in R$, $e \in \{+,-,0\}$, we add to R'_1 the rule $< u,i,h >< e,i,h > \to < v,p(i,h) >< \delta,i,h >$ $\{\overline{= \delta_{all}} \wedge \overline{= \gamma_{all}}\}$; all the objects corresponding to those from the compartment of index i and label h must be moved to the parent compartment - this will happen in the presence of (δ,i,h) when no other transformation will take place; this is obtained by using in R'_1 rules $< a,i,h > \to < a,p(i,h) >$ $\{=< \delta,i,h >\}$, $a \in O$ and $< \delta,i,h > \to \lambda$; the set $(I \times H)_c$ will change now by removing the pair (i,h) from it.

(d.2) When $p(i,h) = (i_0,i_0)$, then the rules above will become $< u,i,h >< e,i,h > \to < v,i_0,i_0 >< \delta,i,h > (v,2)$ $\{\overline{= \delta_{all}} \wedge \overline{= \gamma_{all}}\}$ and $< a,i,h > \to < a,i_0,i_0 > (a,2)$ $\{= (\delta,i,h)\}$, $a \in O$.

(e) For each $(i,h) \in I \times H \setminus \{(i_0,i_0)\}$ and each rule $[u]_h^{e_1} \to [v]_h^{e_2}[w]_h^{e_3} \in R$, $e_1,e_2,e_3 \in \{+,-,0\}$; we add to R'_1 the rule $< u,i,h >< e_1,i,h > \to < v,j_1,h >< e_2,j_1,h >< w,j_2,h >< e_3,j_2,h >< \delta,i,h >$ $\{\overline{= \delta_{all}} \wedge \overline{= \gamma_{all}}\}$ – the pair (i,h) is removed from $(I \times H)_c$ and two new pairs (j_1,h) and (j_2,h), existing in $I \times H$, with $j_1 \neq j_2$, are added to $(I \times H)_c$ and one $< u,i,h >$ is transformed into $< v,j_1,h >$ and $< w,j_2,h >$ and their associated electrical charges; then the content corresponding to compartment of index i and label h will be moved to those of index j_1 and j_2 and the same label h, hence rules $< a,i,h > \to < a,j_1,h >< a,j_2,h >$ $\{=< \delta,i,h >\}$, $a \in O$ are added to R'_1; finally, $< \delta,i,h > \to \lambda$ is also included in the set of rules of C_1; it is clear that only one division rule for the same compartment is applied in any step of the computation.

We note that in C'_2 there are no rules for dissolution and division as the output compartment is not afected by these rules.

The execution strategy in both compartments, C_1 and C_2 is maximal parallelism.

For a sequence of rules applied in Π, we have a corresponding sequence of rules in $k\Pi$. Obviously the objects obtained in the output compartment of Π are the same with those obtained in C_2 of $k\Pi$.

3.2 P Systems with Symport/Antiport versus kP Systems

The following definition is from [29].

Definition 9. *A P system (of degree $d \geq 1$) with* antiport *and/or* symport *rules is a construct*

$$\Pi = (O, F, E, \mu, w_{1,0}, \cdots, w_{d,0}, R_1, \cdots, R_d, i_0) \quad where$$

*O is the alphabet of objects; $F \subseteq O$ is the alphabet of terminal objects; $E \subseteq O$ is the set of objects occurring in an unbounded number in the environment; μ is a membrane structure consisting of d membranes (usually labelled with i and represented by corresponding brackets $[_i$ and $]_i$, $1 \leq i \leq d$); w_i, $1 \leq i \leq d$, are strings over O associated with regions $1, \cdots, d$ of μ, representing the initial multisets of objects present in the regions of μ; R_i, $1 \leq i \leq d$, are finite sets of rules of the form $(u, out; v; in)$, with $u \neq \lambda$ and $v \neq \lambda$ (*antiport rule*) and/or (x, out) or (x, in), with $x \neq \lambda$ (*symport rules*); i_0, $1 \leq i_0 \leq d$, specifies the output membrane of Π.*

We will show now that one can construct for any symport/antiport P system a kernel P system, such that they compute the same result. We will adopt a slightly different way of computing the result of the kP systems by allowing it to use a set of terminal objects. In this case, according to Remark 4, the result will be given by the number of terminal objects from the output compartment. We can now state the main result of this section.

Theorem 2. *For any P system with symport/antiport rules, Π, there is a kP system, $k\Pi$, using only rewriting and communication rules and having a terminal set of objects, such that $M(\Pi) = M_t(k\Pi)$.*

Proof. Let $\Pi = (O, F, E, \mu, w_{1,0}, \cdots, w_{d,0}, R_1, \cdots, R_d, i_0)$ be a P system, of degree d, with symport and antiport rules as given by Definition 9.

We construct a kP system $k\Pi$ of degree one in the following manner. We take one unique compartment C_1. Apart from the d membranes in system Π, numbered by $1, 2, \cdots, d$, we think of the environment as a new membrane, with label 0.

The kP system we build is $k\Pi = (A, F', \mu', C_1, 1)$. The alphabet, A, of $k\Pi$ will consist of objects given by pairs $< x, i > \in O \times \{0, 1, \cdots, d\}$. For a multiset $w = a_1 \cdots a_m$ in membrane i we use the notation $< w, i >$ for $< a_1, i >, \cdots < a_m, i >$.

The initial multiset is

$$w'_{1,0} =< w_{1,0}, 1 > \cdots < w_{d,0}, d >$$

i.e., it contains all the pairs having the first element the initial multiset of membrane i and the second one i, $1 \leq i \leq d$. Initially, the environment associated with Π does not have any other objects apart from those in E. The set of rules, R'_1, of the kP system, includes the rules below.

- If a rule $(u, out; v, in)$, $u \neq \lambda$, $v \neq \lambda$, is in membrane i with parent j and $j \neq 0$, then we add the rule
  $< u, i >< v, j > \to < u, j >< v, i >$.
- If a rule $(u, out; v, in)$, $u \neq \lambda$, $v \neq \lambda$, is in membrane i with parent $j, j = 0$, then we decompose $u = u_1 u_2$ and $v = v_1 v_2$, such that $u_1, v_1 \in (O \setminus E)^*$ and $u_2, v_2 \in E^*$ and add the rule
  $< u, i >< v_1, 0 > \to < u_1, 0 >< v, i >$.

If $u_1 = \lambda$ or $v_1 = \lambda$ we interpret $< \lambda, 0 >$ as λ, i.e. for $v_1 = \lambda$ and $u_1 \neq \lambda$ the rule becomes $< u, i > \rightarrow < u_1, 0 >< v, i >$.

- If a rule (u, out), $u \neq \lambda$, is in membrane i with parent j and $j \neq 0$, then we add the rule
  $< u, i > \rightarrow < u, j >$.
- If a rule (u, out), , $u \neq \lambda$, is in membrane i with parent $j, j = 0$, we add the rule
  $< u, i > \rightarrow < u_1, 0 >$,
  where $u = u_1 u_2$ with $u_1 \in (O \setminus E)^*$ and $u_2 \in E^*$.
  If $u_1 = \lambda$, then again $< \lambda, 0 >$ is λ, and the rule becomes $< u, i > \rightarrow \lambda$.
- If a rule (v, in), $v \neq \lambda$, is in membrane i with parent j, and $j \neq 0$, then we add the rule
  $< v, j > \rightarrow < v, i >$.
- If a rule (v, in), $v \neq \lambda$, is in membrane i with parent $j, j = 0$, then we add the rule
  $< v_1, 0 > \rightarrow < v, i >$,
  where $v = v_1 v_2$ with $v_1 \in (O \setminus E)^+$ and $v_2 \in E^*$.
  Note that in this last case $v_1 \neq \lambda$.

Note that the environment (membrane 0) is treated differently by the above rules. We do not keep track of elements over E in the environment, which are in an unbounded number, but we must keep track of elements over $O \setminus E$ in the environment. If an u must go into the environment, then we decompose $u = u_1 u_2$ such that $u_1 \in (O \setminus E)^*$ and $u_2 \in E^*$, and only $< u_1, 0 >$ will appear in the right-hand side of the rule. Similarly, if a v comes from the environment, we have $v = v_1 v_2$ with $v_1 \in (O \setminus E)^+$ and $v_2 \in E^*$, and $< v_1, 0 >$ must be consumed by the rule.

The execution strategy of $k\Pi$ will be maximal parallelism.

The terminal alphabet is $F' = \{< a, i_0 > \mid a \in F\}$. Note that multisets over F' obtained in $k\Pi$ will correspond to multisets over F obtained in membrane i_0 by Π.

Remark 5. It remains an open problem to devise a kP system with two compartments, where C_1 reflects the functioning of the entire system, while C_2 simulates membrane i_0.

4 Sorting with kP Systems

Sorting is a central topic in Computer Science (see [25]). A variety of approaches to sorting have been investigated, for different algorithms, and with different P system models. A first approach was [3], in which a BeadSort algorithm was implemented with tissue P systems. Another approach was [6], in which algorithms inspired from sorting networks were implemented using P systems with communication. Other papers ([1], [30]) use different types of P systems, and refine the sorting problem to sorting by ranking. A first overview of sorting algorithms implemented with P systems was [2]. A dynamic sorting algorithm was proposed in [7]. The bitonic sort was implemented with P systems [8], spiking neural P systems were used for sorting [10], other network algorithms were implemented using P systems [9]. Another overview of sorting algorithms implemented with P systems is provided by [11]. First implementations of sorting with kP systems were proposed in [16, 17].

The problem can be stated as follows: suppose we want to sort $x_1, \cdots, x_n$, $n \geq 1$, in ascending order, where x_i, $1 \leq i \leq n$, are positive integer values. Each such number, x_i, $1 \leq i \leq n$, will be represented as a multiset $a_i^{x_i}$, $1 \leq i \leq n$, where a_i is an object from a given set. In the next sections we will present two sorting algorithms using different representations of the sequence of positive integer numbers. More precisely, we start with an algorithm already studied in several other papers, [6, 2] for various types of P systems. Here we implement it using kP systems, by representing each element x_i by a^{x_i}, $1 \leq i \leq n$. The multisets a^{x_i}, $1 \leq i \leq n$, are stored in separate compartments, C_i, $1 \leq i \leq n$ (Section 4.1). In Section 4.2 these positive integer numbers are represented by $a_i^{x_i}$, $1 \leq i \leq n$, and stored in one compartment C_1; an additional one, C_2, is used for implementation purposes. In Section 4.3 it is used again the representation $a_i^{x_i}$, $1 \leq i \leq n$, but a more complex structure of compartments is provided in order to maximise the parallel behaviour of the system implementing the sorting algorithm. The algorithm used in Section 4.1 and Section 4.2 makes comparisons of adjacent compartments by employing a two stage process. In the first stage all pairs "odd-even" are compared (C_{2i-1} with C_{2i}, $i \geq 1$) and in the second stage all pairs "even-odd" are involved (C_{2i} with C_{2i+1}, $i \geq 1$).

4.1 Sorting Using kP Systems with an Element per Compartment

The approach presented below follows [16, 17], but stopping conditions have been also considered and the sequence of numbers is obtained in ascending order.

Let us consider a kP system, $k\Pi_1$, having n compartments $C_i = (t_i, w_{i,0})$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq n$, and a set of objects $A = \{a, b, c, p, p'\}$. In each compartment, C_i, the initial multiset, $w_{i,0}$, $1 \leq i \leq n$, includes the representation of the positive integer number x_i, i.e., a^{x_i}, the multiset $c^{2(n-1)}$ and the object p for all odd index values, when n is an even number, and for all odd index values, but the last, when n is odd. The objects p stored initially in compartments indexed by odd values indicate that one starts with stage one, whereby "odd-even" compartment pairs are compared first. The multiset $c^{2(n-1)}$ will be used in a counting process, in each of the compartments, that will help stoping the algorithm when the sorting is complete.

Let us consider for $n = 6$ the sequence 3, 6, 9, 5, 7, 8. Then the initial multisets are:
$w_{1,0} = a^3c^{10}p; w_{2,0} = a^6c^{10}; w_{3,0} = a^9c^{10}p; w_{4,0} = a^5c^{10}; w_{5,0} = a^7c^{10}p; w_{6,0} = a^8c^{10}$. As n is even, p appears in all compartments indexed by odd values, i.e., C_1, C_3, and C_5.

In each compartment C_i, t_i contains the following set of rules, denoted R_i, $1 \leq i \leq n$,
$r_{1,i} : a \rightarrow (b, i+1)\ \{\geq p\}$, $i < n$;
$r_{2,i} : p \rightarrow p'$;
$r_{3,i} : p' \rightarrow (p, i+1)$, for i odd and $i < n$, and $r'_{3,i} : p' \rightarrow (p, i-1)$, for i even and $i > 1$;
$r_{4,i} : ab \rightarrow a(a, i-1)$, $i > 1$;
$r_{5,i} : b \rightarrow a$, $i > 1$.

We also consider the rule $r : c \rightarrow \lambda$. This rule is used for implementing the counting process mentioned above. By using the two stage process of comparing "odd-even" pair of compartments and then "even-odd" ones, one needs at most $n-1$ stages to complete the sorting. As it will be explained below, each stage will involve two steps and consequently after $2(n-1)$ steps one expects to stop the sorting process.

In each compartment C_i, the execution strategy is given by

$$\sigma_i = \{r\}\{r_{1,i}, r_{2,i}, r_{3,i}, r_{4,i}\}^{\top}\ \{r_{5,i}\}^{\top},$$

if i is odd; for even values of i, $r_{3,i}$ is replaced by $r'_{3,i}$. The execution strategy, σ_i, tells us that a sequence of three sets of rules are executed in each step. The first one indicates that one single rule is applied and then two sets of rules are used, each of them applied in a maximal parallel manner.

We assume that any two compartments, C_i, C_{i+1}, $1 \leq i < n$, are connected.

In the first step, of the "odd-even" stage, in every compartment one c is removed by applying $r : c \rightarrow \lambda$; then the only applicable rules are $r_{1,i}, r_{2,i}$ in all compartments indexed by an odd value. Given the presence of p in these compartments, rules $r_{1,i}$ move all objects a from each compartment with an odd index value, i, $i < n$, to the compartment C_{i+1} by transforming them into bs and rules $r_{2,i}$ transforming p into p'. In the next step, another c is removed from every compartment and rules $r_{3,i}, r_{4,i}, r_{5,i}$ are then applied. The rules $r_{3,i}$ are applied in compartments with an odd index value and $r_{4,i}$ are applied in compartments with an even index value, this means p' is moved as p from each C_i, i an odd value and $i < n$, to compartment C_{i+1} and every ab, in each C_j, j an even value and $j > 1$, is transformed into an a kept in the compartment and another a moved to C_{j-1}. At the end of the step, in each compartment C_j, j an even value and $j > 1$, and in accordance with the execution strategy, the remaining b objects, if any, are transformed into as. These two steps implement comparators between two adjacent compartments, in this case "odd-even" pairs. If a^{x_i} from C_i and $a^{x_{i+1}}$ from C_{i+1}, $i < n$, are such that $x_i > x_{i+1}$ then the multisets a^{x_i} is moved to C_{i+1} and $a^{x_{i+1}}$ to C_i. In the next step, the first of the second stage, ps appear in even compartments and the comparators are now acting between pairs of compartments C_i, C_{i+1}, where i is even and $i < n$.

Given that the algorithm must stop in maximum $2(n-1)$ steps, one can notice that in step $2(n-1)$ the counter, c, disappears, i.e., becomes λ, and the first rule from the execution strategy, r, is no longer applicable and then the next sets of rules are not executed either. Hence, the process stops with the multisets codifying for positive integer values in asceding order.

The table below presents the first four steps of the sorting process.

Compartments - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	$a^3c^{10}p$	a^6c^{10}	$a^9c^{10}p$	a^5c^{10}	$a^7c^{10}p$	a^8c^{10}
1	c^9p'	$a^6b^3c^9$	c^9p'	$a^5b^9c^9$	c^9p'	$a^8b^7c^9$
2	a^3c^8	a^6c^8p	a^5c^8	a^9c^8p	a^7c^8	a^8c^8p
3	a^3c^7	c^7p'	$a^5b^6c^7$	c^7p'	$a^7b^9c^7$	a^8c^7p'
4	a^3c^6p	a^5c^6	a^6c^6p	a^7c^6	a^9c^6p	a^8c^6

Now, one can state the result of the algorithm presented above and the number of steps involved.

Theorem 3. *The above algorithm sorts in ascending order a sequence of* $n, n \geq 1$, *positive integer numbers in* $2(n-1)$ *steps.*

4.2 Sorting Using kP Systems with Two Compartments

In this section we use a representation of the positive integer numbers $x_1, \cdots, x_n$ as multisets $a_1^{x_1}, \cdots, a_n^{x_n}$, where $a_1, \cdots, a_n$ are from a given set of objects. We consider a kP system, $k\Pi_2$, with two compartments $C_j = (t_j, w_{j,0})$, $1 \leq j \leq 2$, which are linked and $A = \{a_1, \ldots, a_n, c\}$. The initial multisets are $w_{1,0} = a_1^{x_1} \cdots a_n^{x_n} c^{n-1}$ and $w_{2,0} = c^{n-1}$.

Finally, the kP system $k\Pi_2$ will lead to a multiset $a_1^{x_{i_1}} \cdots a_n^{x_{i_n}}$ in compartment C_1, such that $x_{i_1} \leq \cdots \leq x_{i_n}$.

In compartments C_1 the rules are
$R_{1,1} = \{a_i a_{i+1} \to (a_i, 2)(a_{i+1}, 2) \mid 1 \leq i < n \;\&\; i = 1, 3, ...\}$;
$R_{2,1} = \{a_i \to (a_{i+1}, 2) \mid 1 \leq i < n \;\&\; i = 1, 3, ...\}$;
$R_{3,1} = \{a_i \to (a_i, 2) \mid 1 \leq i \leq n\}$.

We also consider the rule $r : c \to \lambda$, like in the previous section.

Compartment C_2 has the rules
$R_{1,2} = \{a_i a_{i+1} \to (a_i, 1)(a_{i+1}, 1) \mid 1 \leq i < n \;\&\; i = 2, 4, ...\}$;
$R_{2,2} = \{a_i \to (a_{i+1}, 1) \mid 1 \leq i < n \;\&\; i = 2, 4, ...\}$;
$R_{3,2} = \{a_i \to (a_i, 1) \mid 1 \leq i \leq n\}$;
and the rule r defined above.

The execution strategies of these compartments are

$$\sigma_j = \{r\} Lab(R_{1,j})^\top Lab(R_{2,j})^\top Lab(R_{3,j})^\top,\ j = 1, 2.$$

In compartment C_1 one implements "odd-even" comparison steps and in C_2 "even-odd" steps. The process starts with compartment C_1. The execution strategy in each compartment starts by decrementing the counter (using r), then the comparators are implemented by executing first $R_{1,j}$ and then $R_{2,j}$, $j = 1, 2$, both in maximally parallel manner. After that all the pairs a_i, a_{i+1} are sent to the other compartment and when $a_i^{x_i}$ and $a_{i+1}^{x_{i+1}}$ are such that $x_i > x_{i+1}$ then a_i is transformed into a_{i+1} and sent to the other compratment, i.e., a_i and a_{i+1} are swapped and sent to the other compartment. In the last part, are moved to the other compartment all the objects a_i, $1 \leq i \leq n$, that remained there after comparisons. This is the case when a pair a_i and a_{i+1} has its objects with their multiplicities, x_i and x_{i+1}, respectively, in the right order, i.e., $x_i \leq x_{i+1}$.

Clearly after at most $n-1$ steps the objects $a_1, \cdots, a_n$ have their multiplicities in the ascending order and the sorting process stops as r is no longer applicable and the execution strategy is not applicable any more.

Theorem 4. *The above algorithm sorts in ascending order a sequence of $n, n \geq 1$, positive integer numbers in $n-1$ steps.*

One can produce a similar implementation whereby the comparison of two neighbours is made more directly and with simpler rules, but with more complex guards.

In this case we extend the definition of a guard, by allowing θa^n to be of the form $\theta a^{f(z)}$, where $f(z)$ is a function over the multisets of objects returning a positive integer value. For the current multiset z, one can define, for instance, $f_b(z) = |z|_b$, Then a rule $a \to b\{> a^{f_b(\cdot)}\}$ is applicable to z if the guard is true, i.e., $|z|_a > |z|_b$.

The *extended definition of the guard* allows us to implement a comparator with simpler rules than in the previous case. We have the pair of integers x_1, x_2 represented as ${a_1}^{x_1}, {a_2}^{x_2}$. Consider the pair of guarded rewriting rules

$$a_1 \to a_2\{> {a_1}^{f_{a_2}(\cdot)}\} \;\text{ and }\; a_2 \to a_1\{< {a_2}^{f_{a_1}(\cdot)}\}$$

where $f_{a_2}(w) = |w|_{a_2}$ and $f_{a_1}(w) = |w|_{a_1}$. Then both guards codify the condition $x_1 > x_2$.

If $x_1 \leq x_2$ the rules are not applicable, while if $x_1 > x_2$, then the x_1 copies of a_1 are rewritten as a_2, and x_2 copies of a_2 are rewritten as a_1, interchanging the values and achieving eventually $x_1 \leq x_2$.

A kP system, $k\Pi_3$, is defined now for sorting the sequence of $n, n \geq 1$, positive integer numbers. It consists of two compartment C_1 and C_2 which are linked. They have the same initial multisets like $k\Pi_2$. The sets of rules associated with these compartments are

- R_1 consisting of three subsets of rules (R_1 is responsible for "odd-even" stages):
  - $\{r \mid r : c \to \lambda\}$;
  - $R_{1,1} = \{a_i \to (a_{i+1}, 2)\{> a_i^{f_{a_{i+1}}(\cdot)}\} \mid i = 1, 3 \cdots \ \& \ i < n\}$;
  - $R_{2,1} = \{a_{i+1} \to (a_i, 2)\{< a_{i+1}^{f_{a_i}(\cdot)}\} \mid i = 1, 3 \cdots \ \& \ i < n\}$;
  - $R_{3,1} = \{a_i \to (a_i, 2) \mid i = 1, \cdots, n\}$.

The function f_{a_i} is defined $f_{a_i}(z) = |z|_{a_i}$, $1 \leq i \leq n$, for any multiset z.

Similarly, one defines R_2 in compartment C_2, which is used to implement the "even-odd" stage. The execution strategy is given by $\sigma_j = \{r\}Lab(R_{1,j} \cup R_{2,j})^\top Lab(R_{3,j})^\top, j = 1, 2$.

Theorem 5. *The above algorithm sorts in ascending order a sequence of $n, n \geq 1$, positive integer numbers in $n-1$ steps.*

Remark 6. 1. The kP system $k\Pi_3$ has simpler rules (non-cooperative) than $k\Pi_2$ (cooperative rules), but the guards of the rules in $k\Pi_2$ are simpler than those belonging to $k\Pi_3$.
2. The number of rules applied in each step to interchange $a_i^{x_i}$ and $a_{i+1}^{x_{i+1}}$ is $max\{x_i, x_{i+1}\}$ for $k\Pi_2$ and $x_i + x_{i+1}$ for $k\Pi_3$. Hence, $k\Pi_2$ uses less rules than $k\Pi_3$ in each one of the $n-1$ steps.

4.3 A kP System for Sorting in Constant Time

We suppose the integers to be sorted $x_1, \cdots x_n$ distinct.

We use a total of $n^2 + 2n$ compartments:

- $C_{i,j}$, $1 \leq i, j \leq n$, where each $C_{i,j}$ will be responsible for a comparison;
- C_i, $1 \leq i \leq 2n$, where each C_i, $1 \leq i \leq n$, will collect the results of comparing x_i to the rest; and C_i, $n+1 \leq i \leq 2n$, will collect the sorted result.

The connections between compartments are given by the set of edges

$$E = \cup_{i=1}^{n} E_i$$

where

$$E_i = \{(C_i, C_{i,j}) \mid 1 \leq j \leq n\} \cup \{(C_i, C_k) \mid n+1 \leq k \leq 2n\}, 1 \leq i \leq n.$$

Each $C_{i,j}$, $1 \leq i, j \leq n$, will contain the initial multiset $w_{i,j,0} = {a_i}^{x_i} {a_j}^{x_j} a$ and the rules
$r'_{i,j} : a_i \to a_j F\{> {a_i}^{f_j(\cdot)}\}$; $r''_{i,j} : a_j \to a_i\{< {a_j}^{f_i(\cdot)}\}$; $r'''_{i,j} : a \to a'$;
$r_{i,j} : a' \to (F, i)\{\geq F\}$,

where $f_i(z) = |z|_{a_i}$ and $f_j(z) = |z|_{a_j}$.

The execution strategy is $\sigma_{i,j} = \{r'_{i,j}, r''_{i,j}, r'''_{i,j}, r_{i,j}\}^{\top}$.

Note that the rules $r'_{i,j}, r''_{i,j}$ implement a comparator between x_i and x_j, similar to the one of the previous section. The modified comparator produces also a symbol F (False) when $x_i > x_j$, signifying that $x_i \leq x_j$ is false. If the rewriting rules $r'_{i,j}, r''_{i,j}$ and $r'''_{i,j}$ have acted, then a single F will be sent to compartment C_i (by using the rule $r_{i,j}$).

In compartment $C_i, 1 \leq i \leq n$, we have the initial multiset $w_{i,0} = {a_i}^{x_i} a$ and the rules
$r'_i : a \to a'$; $r''_i : a' \to a''$;
$r_{i,0} : a_i \to (a, n+1)\{< F \wedge = a''\}$; $r_{i,k} : a_i \to (a, n+k+1)\{= F^k \wedge = a''\}$, $1 \leq k \leq n-1$.

The execution strategy is $\sigma_i = \{r'_i, r''_i, r_{i,0}, \cdots, r_{i,n-1}\}^{\top}$.

Compartments $C_i, n+1 \leq i \leq 2n$, are initially empty and contain no rules.

The functioning of the system is as follows. Initially, in compartments $C_{i,j}$, $1 \leq i, j \leq n$, the rules $r'_{i,j}$, $r''_{i,j}$, and $r'''_{i,j}$ act. If $x_i > x_j$ the values will be interchanged and some Fs will be produced (rules $r'_{i,j}$, $r''_{i,j}$ are used), signifying that $x_i \leq x_j$ is false. Also $r'''_{i,j}$ is used to transform a in a'. If at least one F is produced in $C_{i,j}$, then a single F will be sent to C_i, using rule $r_{i,j}$. In parallel, in each compartment C_i, $1 \leq i \leq n$, in the first two steps the rules r'_i and r''_i are applied.

After these two steps, no rules are applicable in $C_{i,j}$, $1 \leq i, j \leq n$, and in C_i, $1 \leq i \leq n$, the rules $r_{i,k}$, $0 \leq k \leq n-1$, might be applicable, depending on the number of Fs collected.The number of Fs tells us how many comparisons $x_i \leq x_j$, $1 \leq j \leq n$, are false. If we have k such Fs in C_i, it means that x_i is greater than exactly k other values, which means that in the sorted order it must be the $(k+1)$-th component. This is accomplished by sending a^{x_i} in C_{n+k+1}. The maximum number of Fs in C_i is $n-1$ because $C_{i,i}$ will never produce an F. If there are no Fs in C_i, this means that x_i is the minimum, and a^{x_i} will be sent to C_{n+1}. Compartments C_{n+i}, $1 \leq i \leq n$, collect the result of sorting. Each such C_{n+i} will contain at the end of the computation the string $a^{x_{k_i}}$, x_{k_i} being the i-th value in the sorted order. The computation has three steps, the first two ones in which $C_{i,j}$, $1 \leq i, j \leq n$, work, and a third one in which C_i, $1 \leq i \leq n$, work.

Theorem 6. *The above kP system sorts n integers in 3 steps.*

4.4 Sorting in Constant Time with Membrane Division

The algorithm in the previous section uses only rewriting and communication rules. This solution, although computationally efficient, requires an initial, quite complex, arrangement of compartments and multisets. We present here an algorithm which creates its working space by using membrane division rules.

We want to sort n distinct integers, $x_1, \cdots x_n$, represented as ${a_1}^{x_1}, \cdots, {a_n}^{x_n}$.

We start with a total of $3n$ compartments:

- C_i, $1 \leq i \leq n$, where each C_i, will collect the results of comparing x_i to the rest;
- C_i, $n+1 \leq i \leq 2n$, will collect the sorted result;
- C_k, $2n+1 \leq k \leq 3n$, such that $C_{2n+i} \subset C_i$, responsible for creating the comparator compartments.

The connections between compartments are given by the set of edges

$$E = \cup_{i=1}^{n} E_i$$

where

$$E_i = \{(C_i, C_{2n+i})\} \cup \{(C_i, C_k) \mid n+1 \leq k \leq 2n\}, 1 \leq i \leq n.$$

In compartment $C_i, 1 \leq i \leq n$, we have the initial multiset $w_{i,0} = {a_i}^{x_i} a$ and the rules
$r'_i : a \to a'$; $r''_i : a' \to a''$; $r'''_i : a'' \to a'''$;
$r_{i,0} : a_i \to (a, n+1)\{< F \wedge = a'''\}$; $r_{i,k} : a_i \to (a, n+k+1)\{= F^k \wedge = a'''\}$, $1 \leq k \leq n-1$.

The execution strategy is $\sigma_i = \{r'_i, r''_i, r'''_i, r_{i,0}, \cdots, r_{i,n-1}\}^{\top}$.

Compartments $C_i, n+1 \leq i \leq 2n$, are initially empty and contain no rules.

Compartments C_{2n+i}, $1 \leq i \leq n$, contain an initial multiset s, where s is a new object, and the membrane division rules

$$[s]_{2n+i} \rightarrow [{a_1}^{x_1}{a_i}^{x_i}a]_{i,1} \cdots [{a_j}^{x_j}{a_i}^{x_i}a]_{i,j} \cdots [{a_n}^{x_n}{a_i}^{x_i}a]_{i,n},\ \ 1 \leq i \leq n.$$

These rules will generate in each C_i the compartments $C_{i,j}$, $1 \leq j \leq n$. In each $C_{i,j}$ we will have the multiset ${a_j}^{x_j}{a_i}^{x_i}a$, and the rules

$r'_{i,j} : a_i \rightarrow a_j F\{> {a_i}^{f_j(\cdot)}\}$; $r''_{i,j} : a_j \rightarrow a_i\{< {a_j}^{f_i(\cdot)}\}$; $r'''_{i,j} : a \rightarrow a'$;
$r_{i,j} : a' \rightarrow (F, i)\{\geq F\}$,

where $f_i(z) = |z|_{a_i}$ and $f_j(z) = |z|_{a_j}$.

The execution strategy is $\sigma_{i,j} = \{r'_{i,j}, r''_{i,j}, r'''_{i,j}, r_{i,j}\}^{\top}$.

Note that this is the comparator of the previous section, which sends a single F in C_i if $x_i \leq x_j$ is false.

During the first step, in compartment C_i rule r'_i is executed, while in C_{2n+i} the membrane division rule is applied, generating the $C_{i,j}$, $1 \leq j \leq n$. The next three steps are identical to the ones of the previous algorithm.

Theorem 7. *The above kP system sorts n integers in 4 steps.*

5 Simulating and Verifying kP Systems

In Section 4, we have illustrated that kP systems provide a coherent and expressive language that allow us to model various systems that were originally implemented by different P system variants. In addition to the modelling aspect, there has been a significant progress on analysing kP systems using various simulation and verification methodologies. The methods and tools developed in this respect have been integrated into a software platform, called kPWorkbench, to support the modelling and analysis of kP systems.

The ability of simulating kernel P systems is an important feature of this tool. Currently, there are two different simulation approaches, kPWorkbench Simulator and Flame (Flexible Large-Scale Agent Modelling Environment). Both simulators receive as input a kP system model written in kP–Lingua and outputs a trace of the execution, which is mainly used for checking the evolution of a system and for extracting various results out of the simulation. The simulators provide traces of execution for a kP system model, and an interface displaying the current configuration (the content of each compartment) at each step. It is useful for checking the temporal evolution of a kP system and for inferring various information from the simulation results.

Another important analysis method that kPWorkbench features is formal verification, requiring an exhaustive analysis of system models against some queries to be verified. The automatic verification of kP systems brings in some challenges as they feature a dynamic structure by preserving the structure changing rules such as membrane division, dissolution and link creation/destruction. kPWorkbench

Prop.	Pattern	(i) Informal query, (ii) Formal query using patterns
1	Existence	**(i)** *The numbers will be eventually sorted, i.e. the multisets representing the numbers will be in ascending order in the compartments*
		(ii) eventually (c_1.a <= c_2.a & c_2.a <= c_3.a & c_3.a <= c_4.a & c_4.a <= c_5.a & c_5.a <= c_6.a)
2	Universality	**(i)** *Counters in different compartments are always sync'ed*
		(ii) always (c_1.c = c_2.c & c_2.c = c_3.c & c_3.c = c_4.c & c_4.c = c_5.c & c_5.c = c_6.c)
3	Steady-state	**(i)** *In the state-state, the numbers are sorted*
		(ii) steady-state (c_1.a <= c_2.a & c_2.a <= c_3.a & c_3.a <= c_4.a & c_4.a <= c_5.a & c_5.a <= c_6.a)
4	Existence	**(i)** *The algorithm will eventually stop*
		(ii) eventually (c_i.c = 0)
5	Response	**(i)** *An unsorted state of two adjacent compartments will always be followed by a sorted one*
		(ii) (c_i.a >c_{i+1}.a) **followed-by** (c_i.a <= c_{i+1}.a)

Table 1: List of properties derived from the property language and their representations in different formats.

employs different verification strategies to allieviate these issues. The framework supports both *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)* properties by making use of the Spin [22] and NuSMV [13] model checkers.

In order to facilitate the formal specification, kPWorkbench features a property language, called *kP-Queries*, comprising a list of natural language statements representing formal property patterns, from which the formal syntax of the Spin and NuSMV formulas are automatically generated. The property language editor interacts with the kP-Lingua model in question and allows users to directly access the native elements in the model, which results in less verbose and shorter state expressions, and hence more comprehensible formulas. *kP-Queries* also features a grammar for the most common property patterns. These features and the natural language like syntax of the language make the property construction much easier.

Some of the commonly used patterns are "next", "existence", "absence", "universality", "recurrence", "steady-state", "until", "response" and "precedence". The details can be found in [21].

We now illustrate the usage of the query patters on the sorting algorithm given in Section 4.1. The other algorithms can be considered in a similar manner. In order to verify that the algorithm works as desired, we have constructed a set of properties specified in kP-Queries, listed in Table 1. The applied pattern types are given in the second column of the table. For each property we provide the following information; **(i)** informal description of each kP-Query, and **(ii)** the formal kP-Query using the patterns. The queries given in Table 1 capture that the algorithm given in Section 4.1 works as desired.

We note that both kP–Lingua model and the queries are automatically converted into the languages required by the corresponding model checkers. So, the verification process in kPWorkbench is carried out in automatic manner.

6 Testing kP Systems Using Automata Based Techniques

In this section we outline how the kP systems obtained in the previous sections can be tested using automata based testing methods. The approach presented here follows the blueprint presented in [24] and [15] for cell-like P systems. We illustrate our approach on $k\Pi_1$, the application of our approach on the other kP system modeling sorting algorithms is similar.

Naturally, in order to apply an automata based testing method to a kP model, a finite automata needs to be obtained first. In general, the computation of a kP system cannot be fully modelled by a finite automaton and so an *approximate* automaton will be sought. The problem will be addressed in two steps.

- Firstly, the computation tree of a P system will be represented as a deterministic finite automaton. In order to guarantee the finiteness of this process, an upper bound k on the length of any computation will be set and only computations of maximum k transitions will be considered at a time.
- Secondly, a *minimal* model, that preserves the required behaviour, will be defined on the basis of the aforementioned derivation tree.

Let $M_k = (A_k, Q_k, q_{0,k}, F_k, h_k)$ be the finite automaton representation of the computation tree, where A_k is the finite input alphabet, Q_k is the finite set of states, $q_{0,k} \in Q_k$ is the initial state, $F_k \subseteq Q_k$ is the set of final states, and $h_k : Q_k \times A_k \longrightarrow Q_k$ is the next-state function. A_k is composed of the tuples of multisets that label the transition of the computation tree. The states of T_k correspond to the nodes of the tree. For testing purposes we will consider all the states as final. It is implicitly assumed that a non-final "sink" state q_{sink} that receives all "rejected" transitions, also exists.

Consider $k\Pi_1$, the kP system in section 4.1, $n = 6$ and the sequence to be sorted 3, 6, 9, 5, 7, 8. Then the initial multisets are:
$w_{1,0} = a^3c^{10}p; w_{2,0} = a^6c^{10}; w_{3,0} = a^9c^{10}p; w_{4,0} = a^5c^{10}; w_{5,0} = a^7c^{10}p; w_{6,0} = a^8c^{10}$. As $k\Pi_1$ is a deterministic kP system, there are no ramification in the computation tree. For $k = 3$, this is represented below.

Compartments - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	$rr^3_{1,1}r_{2,1}$	r	$rr^9_{1,3}r_{2,3}$	r	$rr^7_{1,5}r_{2,5}$	r
1	$rr_{3,1}$	$rr^3_{4,2}$	$rr_{3,3}$	$rr^5_{4,4}r^4_{5,4}$	$rr_{3,5}$	$rr^7_{4,6}$
2	r	$rr^6_{1,2}r_{2,2}$	r	$rr^9_{1,4}r_{2,4}$	r	$rr_{2,6}$
3	r	$rr'_{3,2}$	$rr^5_{1,3}r_{5,3}$	$rr'_{3,4}$	$rr^7_{1,5}r^2_{5,5}$	$rr'_{3,6}$

Let us denote
$\alpha_1 = (rr^3_{1,1}r_{2,1}, r, rr^9_{1,3}r_{2,3}, r, rr^7_{1,5}r_{2,5}, r)$,
$\alpha_2 = (rr_{3,1}, rr^3_{4,2}, rr_{3,3}, rr^5_{4,4}r^4_{5,4}, rr_{3,5}, rr^7_{4,6})$,
$\alpha_3 = (r, rr^6_{1,2}r_{2,2}, r, rr^9_{1,4}r_{2,4}, r, rr_{2,6})$,
$\alpha_4 = (r, rr'_{3,2}, rr^5_{1,3}r_{5,3}, rr'_{3,4}, rr^7_{1,5}r^2_{5,5}, rr'_{3,6})$.

Then, for $k = 3$, $M_k = (A_k, Q_k, q_{0,k}, F_k, h_k)$, where
$A_k = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$, $Q_k = \{q_{0,k}, q_{1,k}, q_{2,k}, q_{3,k}, q_{4,k}\}$, $F_k = Q_k$, and h_k, the next-state function, is defined by: $h_k(q_{i-1,k}, \alpha_i) = q_{i,k})$, $1 \leq i \leq 4$.

As M_k is a deterministic finite automaton over A_k, one can find the minimal deterministic finite automaton that accepts *exactly* the language defined by M_k. However, as only sequences of at most k transitions are considered, it is irrelevant how the constructed automaton will behave for longer sequences. Consequently, a deterministic finite cover automaton of the language defined by M_k will be sufficient.

A *deterministic finite cover automaton* (*DFCA*) of a finite language U is a deterministic finite automaton that accepts all sequences in U and possibly other sequences that are longer than any sequence in U [4], [5]. A *minimal DFCA* of U is a DFCA of U having the least possible states. A minimal DFCA may not be unique (up to a renaming of its states). The great advantage of using a minimal DFCA instead of the minimal deterministic automaton that accepts precisely the language U is that the size (number of states) of the minimal DFCA may be much less than that of the minimal deterministic automaton that accepts U. Several algorithms for constructing a minimal DFCA (starting from the deterministic automaton that accepts the language U) exist, the best known algorithm [26] requires $O(n\ log\ n)$ time, where n denotes the number of states of the original automaton. For details about the construction of a minimal DFCA we refer the reader to [24] and [26].

A minimal DFCA of the language defined by M_k, $k = 3$, is $M = (A, Q, q_0, F, h)$, where $A = A_k$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = Q$ and h defined by: $h(q_{i-1}, \alpha_i) = q_i$, $1 \leq i \leq 3$ and $h(q_3, \alpha_4) = q_0$.

Now, suppose we have a finite state model (automaton) of the system we want to test. In *conformance testing* one constructs a finite set of input sequences, called *test suite*, such that the implementation passes all tests in the test suite if and only if it behaves identically to the specification on any input sequence. Naturally, the implementation under test can also be modelled by an unknown deterministic finite automaton, say M'. This is not known, but one can make assumptions about it (e.g. that may have a number of incorrect transitions, missing or extra states). One of the least restrictive assumptions refers to its size (number of states). The *W-method* [12] assumes that the difference between the number of states of the implementation model and that of the specification has to be at most β, a non-negative integer estimated by the tester. The W-method involves the selection of two sets of input sequences, a state cover S and a characterization set W [12].

In our case, we have constructed a DFCA model of the system and we are only interested of the behavior of the system for sequences of length up to an upper bound k. Then, the set suite will only contain sequences of up to length k and its successful application to the implementation under test will establish that the implementation will behave identically to the specification for any sequence of length less then or equal to k. This situation is called conformance testing for bounded sequences. Recently, it was shown that the underlying idea of the W-method can also be applied in the case of bounded sequences, provided that the sets S and W used in the construction of the test suite satisfy some further requirements; these are called a proper state cover and strong characterization set, respectively [23]. In what follows we informally define these two concepts and illustrate them on our working example. For formal definitions we refer the reader to [23] or [24].

A *proper state cover* of a deterministic finite automaton $M = (A, Q, q_0, F, h)$ is a set of sequences $S \subseteq A^*$ such that for every state $q \in Q$, S contains a sequence of *minimum length* that reaches q. Consider M the DFCA in our example. Then λ is the sequence of minimum length that reaches q_0, σ_1 is a sequence of minimum length that reaches q_1, $\alpha_1\alpha_2$ is a sequence of minimum length that reaches q_2, $\alpha_1\alpha_2\alpha_3$ is a sequence of minimum length that reaches q_3. Furthermore, we can use any input symbol in $A \setminus \{\alpha_1\}$ to reach the (implicit) "sink" state, for example α_2. Thus, $S = \{\lambda, \alpha_1, \alpha_1\alpha_2, \alpha_1\alpha_2\alpha_3, \alpha_2\}$ is a proper state cover of M.

A *strong characterization set* of a minimal deterministic finite automaton $M = (A, Q, q_0, F, h)$ is a set of sequences $W \subseteq A^*$ such that for every two distinct states $q_1, q_2 \in Q$, W contains a sequence of minimum length that distinguishes between q_1 and q_2. Consider again our running example. λ distinguishes between the (non-final) "sink" state and all the other (final) states. A transition labelled α_1 is defined from q_0, but not from q_1, q_2 or q_3, so α_1 is a sequence of minimum length that distinguishes q_0 from q_1, q_2 and q_3. Similarly, α_2 is a sequence of minimum length that distinguishes q_1 from q_2 and q_3 and α_3 is a sequence of minimum length that distinguishes between q_2 and q_3. Thus $W = \{\lambda, \alpha_1, \alpha_2, \alpha_3\}$ is a strong characterization set of M,

Once we have established the sets S and W and the maximum number β of extra states that the implementation under test may have, a test suite is constructed by extracting all sequences of length up to k from the set

$$S(A^0 \cup A^1 \cup \ldots \cup A^\beta)W,$$

where A^i denotes the set of input sequences of length $i \geq 0$.

Note that some test sequences may be accepted by the DFCA model - these are called *positive tests* - but some others may not be accepted (they end up in the (non-final) "sink" state) - these are called *negative tests*.

7 Conclusions

In this paper, we have investigated the relationships between kP systems, on the one hand, and active membrane systems with polarization and symport/antiport membrane systems, on the other hand. We have also illustrated the modeling power of kP systems by providing a number of kP system models for sorting algorithms. We have also discussed the problem of testing systems modelled as kernel P systems and proposed a test generation method based on automata. Namely, we have outlined how the kP systems can be tested using automata based testing methods. Furthermore, we have shown how formal verification can be used to validate that the given models work as desired.

We have also begun a study on the ability of kP aytems to simulate other particular classes of P systems, We have presented here the case of P systems with active membranes, and P systems with symport/antiport rules.

In future studies we aim to connect kP systems with other classes of P systems, especially those utilised in various applications, and to show how other problems can be solved, tested and verified by using kP systems.

Acknowledgements.

MG and SK acknowledge the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1). The work of FI and MG were supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688).

References

1. A. Alhazov, D. Sburlan, Static Sorting Algorithms for P Systems, *Pre-Proc. 4th Workshop on Membrane Computing* (A. Alhazov et al., eds.), *GRLMC Rep.* 28/03, Tarragona, 17 – 40, 2003.
2. A. Alhazov, D. Sburlan, Static Sorting P Systems. In [14], 215 – 252, 2006.
3. J.J. Arulanandham, Implementing Bead-Sort with P Systems, *Unconventional Models of Computation* (C.S. Calude et al., eds.), *Lecture Notes in Computer Science*, 2509, 115–125, 2002.
4. C. Câmpeanu, N. Santean, S. Yu. Minimal Cover-Automata for Finite Languages, *Workshop on Implementing Automata* (J.-M. Champarnaud et al., eds.), *Lecture Notes in Computer Science*, 1660, 43 – 56, 1998.
5. C. Câmpeanu, N. Santean, S. Yu. Minimal Cover-Automata for Finite Languages. *Theoretical Computer Science*, 267(1-2), 3 – 16, 2001.
6. R. Ceterchi, C. Martín-Vide, P Systems with Communication for Static Sorting. In *Pre-Proc. 1st Brainstorming Week on Membrane Computing* (M. Cavaliere et al., eds.), *Technical Report* no 26, Rovira i Virgili Univ., Tarragona, 101 – 117, 2003.
7. R. Ceterchi , C. Martín-Vide, Dynamic P Systems, *Proc. 4th Workshop on Membrane Computing* (Gh. Păun et al., eds.), *Lecture Notes in Computer Science*, 2597, 146 – 186, 2003.
8. R. Ceterchi, M.J. Pérez-Jiménez, A.I. Tomescu, Simulating the Bitonic Sort Using P Systems, *Proc. 8th Workshop on Membrane Computing* (G. Eleftherakis et al., eds.), *Lecture Notes in Computer Science*, 4860, 172 – 192, 2007.
9. R. Ceterchi, M.J. Pérez-Jiménez, A.I. Tomescu, Sorting Omega Networks Simulated With P Systems: Optimal Data Layouts, (D. Diaz-Pernil et al., eds.), *Pre-Proc. 6th Brainstorming Week on Membrane Computing*, *RGNC Rep.* 01/08, Fénix Editora, pp. 79 – 92, 2008.
10. R. Ceterchi, A. I. Tomescu, Implementing Sorting Networks with Spiking Neural P Systems, *Fundamenta Informaticae*, 87(1), 35 – 48, 2008.

11. R. Ceterchi, D. Sburlan, Membrane Computing and Computer Science, Chapter 22 of [29], 553–583, 2010.
12. T. S. Chow Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3), 178 – 187, 1978.
13. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV Version 2: An Open Source Tool for Symbolic Model Checking, *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, (W.A. Hunt, Jr and F. Somenzi, eds.), *Lecture Notes in Computer Science*, 2404, 359 – 364, 2002.
14. G. Ciobanu, Gh. Păun, M. J. Pérez-Jiménez, eds., *Applications of Membrane Computing*, Springer, 2006.
15. M. Gheorghe, F. Ipate. On Testing P Systems, *Proc. 9th Workshop on Membrane Computing*, (D.W. Corne et al., eds.), *Lecture Notes in Computer Science*, 5391, 204 – 216, 2009.
16. M. Gheorghe, F. Ipate, C. Dragomir, Kernel P Systems, *Pre-proc. 10th Brainstorming Week on Membrane Computing*, (M. A. Martínez-del-Amor et al., eds.), Fénix Editora, Universidad de Sevilla, 153 – 170, 2012.
17. M. Gheorghe, F. Ipate, C. Dragomir, L. Mierlă, L. Valencia-Cabrera, M. García-Quismondo, M.J. Pérez-Jiménez, Kernel P Systems – Version 1, *Pre-Proc. 11th Brainstorming Week on Membrane Computing*, (L. Valencia-Cabrera et al., eds.), Fénix Editora, Universidad de Sevilla, 97 – 124, 2013.
18. M. Gheorghe, F. Ipate, S. Konur, Kernel P Systems and Relationships with other Classes of P Systems, *Multidisciplinary Creativity*, (M. Gheorghe et al., eds.), Spandugino Publishing House, 64 – 76, 2015.
19. M. Gheorghe, F. Ipate, R. Lefticaru, M.J. Pérez-Jiménez, A. Ţurcanu, L. Valencia-Cabrera, M. García-Quismondo, L. Mierlă, 3-COL Problem Modelling Using Simple kernel P Systems, *International Journal of Computer Mathematics*, 90(4), 816 – 830, 2013.
20. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, Research Frontiers of Membrane Computing: Open Problems and Research Topics, *International Journal of Foundations of Computer Scienve*, 24, 547 – 624, 2013.
21. M. Gheorghe, S. Konur, F. Ipate, L. Mierlă, M. E. Bakir, M. Stannett, An Integrated Model Checking Toolset for Kernel P Systems, *Proc. 16th Conference on Membrane Computing*, (G. Rozenberg et al., eds.), *Lecture Notes in Computer Science*, 9504, 153 – 170, 2015.
22. G. J. Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering*, 23(5), 275 – 295, 1997.
23. F. Ipate, Bounded Sequence Testing from Deterministic Finite State Machines, *Theoretical Computer Science*, 411(16-18), 1770 – 1784, 2010.
24. F. Ipate, M. Gheorghe. Finite State Based Testing of P Systems, *Natural Computing*, 8(4), 833 – 846, 2009.
25. D.E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Addison-Wesley, 1973.
26. H. Körner. On Minimizing Cover Automata for Finite Languages in O(n log n) Time, *Proc. 7th Conference on Implementation and Application of Automata*, (J.-M. Champarnaud and D. Morel, eds.), *Lecture Notes in Computer Science*, 2608, 117 – 127, 2002.
27. Gh. Păun, Computing with Membranes, *Journal of Computer and System Sciences*, 61(1), 108 – 143, 2000.

28. Gh. Păun, Membrane Computing - An Introduction, Springer, 2002.
29. Gh. Păun, G. Rozenberg, A. Salomaa, eds., *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
30. D. Sburlan, A Static Sorting Algorithm for P Systems with Mobile Catalysts, *Analele Ştiinţifice Universitatea Ovidius Constanţa*, 11(1), 195 – 205, 2003.