Specifying Dynamic Software Architectures by Using Membrane Systems

Matteo Cavaliere[1], Vincenzo Deufemia[2]

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `martew@inwind.it`

[2] Dipartimento di Matematica e Informatica
Università di Salerno
Fisciano(SA), Italy
E-mail: `deufemia@unisa.it`

Summary. We present a formalism for the definition of dynamic software architectures in terms of membrane systems, distributed computational models inspired from the structure and the functioning of living cells. The dynamics (the evolution) of the overall architecture is defined by rules that modify the contents (data) and structure of the membrane system. The evolution of the membrane system can be statically checked to ensure that some properties imposed by the architecture are preserved.

1 Introduction

A software architecture provides a high-level system description in terms of a collection of computational components and connectors. Components typically encapsulate information or functionality while connectors coordinate the communication between components. In particular, software architectures describe the decomposition of a system into components, the interconnection of the components, and component interaction [18] allowing developers to abstract away the details of the individual components of an application. Dynamic software architectures are those that change their structure and enact the modifications during the system's execution [10].

In the last years several architecture description languages (ADL) have been proposed to provide behavioral descriptions of the components and connectors [10]. However, to support the development of robust and correct architectures a variety of formal approaches have been introduces (see [11, 3, 6] among others). The importance of having a formal basis for the design of system architectures is underlined in the survey [1].

Besides the possibility of analyzing software architecture properties rigorously (like for instance, deadlock freeness), the formalisms for specifying software architectures are particularly useful for the description of dynamic software systems since their development is "challenging in terms of correctness, robustness, and efficiency" [19]. Indeed, the possibility of instantiating and removing components at run-time makes dynamic software architectures hard to model and analyze since they may have infinite different configurations.

Here, we propose to formally define (dynamic) software architectures in terms of membrane systems, that are parallel and distributed computational devices inspired by the structure and by the working of living cells. For an introduction to this topic, the reader can consult [14] or the guide [15]. The updated literature on this field can be found in the web page [21]. Some similarities with our approach can be found with the approach proposed in [8] where a language based on chemical reactions have been used to formally describe a software system. On the other hand in the approach presented here we introduce many operations typical of the functioning of living cells, that, up to our knowledge, have not been used so far in software system modelling.

The underlying idea of our approach is to represent a certain software architecture by means of a membrane system. Each region of the membrane system represents an abstract component of the software system, with a certain type (indicated by a label taken in a certain alphabet) that defines certain associated rules (operations), used to evolve and to move objects (data). Objects can also have different types (indicated by different symbols of an alphabet) and each occurrence of an object can be involved in certain rules, depending on its type. The objects have also associated information about the membranes they cross during their movements. All the objects (data) are processed and moved in a parallel way.

In this way, the evolution of a membrane system can describe the evolution of a software architecture. Therefore by investigating the representing membrane system, it is possible to formally verify the properties of the modelled software architecture.

2 Preliminaries: Membrane Systems

Membrane systems (also referred to as P systems) are a class of parallel and distributed computing devices inspired from the structure and the functioning of living cells.

The basic model considers a *membrane structure*, consisting of several cell-like membranes which are hierarchically embedded in a main membrane, called the *skin membrane*. The membrane delimits *regions*, where are placed occurrences of *objects* representing chemicals.

The *occurrences of the objects* evolve according to given *evolution rules*, representing the chemical reactions, that are associated to the regions; here we also associate labels to the membranes and we consider rules that can change the membrane structure (in other words, the membranes are considered as *active* elements).

It is also possible to associate rules to the membranes that move the objects from a region to another one (*symport/antiport* rules).

As a general idea, a single step of the evolution of a P system is done by applying, in a non-deterministic maximally parallel manner, the rules associated to the regions and to the membranes of the system.

The evolution of the P system halts when no further rule can be applied to the occurrences of objects and to membranes. A more formal definition of the evolution will be specified later when a specific model of membrane computing will be introduced.

The membrane system model used in this paper is a compound of many (generalized) features taken from the membrane systems area. To give a reference to the current literature, the model used here is a kind of *evolution-communication model* (where evolution rules, together with symport/antiport rules are present) extended with rules of membrane systems using active membranes.

The evolution-communication model has been originally introduced in [4], while membrane systems using active membranes have been investigated in several papers (for instance, [12, 13]). In this paper, we are mainly interested in the dynamics of a membrane system, not in its computational power, and then, we will not specify the output region of the system, contrarily to what is usually done in the membrane computing area.

In this paper we introduce a feature that is not present in any of the membrane models considered up to this moment and that seems necessary for software system modelling, i.e. the association of a unique *identifier* to each membrane. Notice that such an identifier differs from the label of the membrane that indicates the type of the membrane; while several membranes can have the same label (i.e., can have the same type), the identifier of a membrane is unique (i.e., can be imagined as an integer number) and different membranes (even with the same label) have different identifiers; the identifier is associated to a membrane when the membrane is created. Because our goal is to model software systems, we can suppose to take a set of identifiers (ID), large "enough", to be "compatible" with the physical limits (for instance, memory limits) of the software system we plan to model.

In the membrane model defined here, also the occurrences of objects make use of the mentioned identifiers. In fact, each object present in the regions of the membrane system is associated with a string (possibly empty) composed of some of the identifiers of the membranes present in the current configuration.

This string is used *to keep trace*, during the evolution of the system, of the membranes visited by the object occurrence. The rules that move and manipulate the objects can also update the string of identifiers associated to such objects, in a specified way. Some of the rules may be active only when a specified boolean predicate defined over the identifiers is true.

To make more readable the formal definition given later, we introduce some basic operations involving identifiers and strings of identifiers. The hierarchical structure of a P system can be represented by a string of correctly matching parentheses; there is an unique external pair of parentheses representing the skin membrane. We can represent a configuration of a P system adding to such string of parentheses the contents of the regions in form of strings representing multisets.

Given a string μ representing a configuration of a P system, then $\mu(i)$ denotes the ith membrane of μ (i.e., the ith open parenthesis [ ), reading μ as a string, from left to right; $\mu(i).id$ denotes the identifier associated to the ith membrane present in μ. As already mentioned, each membrane of a system has associated a unique identifier. For instance, given $\mu = [_{h_1} ab[_{h_2} c[_{h_1} a]_{h_1} b]_{h_2}]_{h_1}$, then $\mu(3).id$ denotes the identifier associated to the innermost membrane (as mentioned above, to each object occurrence is associated a string of identifiers and such string may even be empty); given a multiset of objects, described by a string m, $m(i)$ denotes the i object occurrence present in the multiset, reading the string m like a string, from left to right; $m(i).list$ denotes the string of identifiers associated to the ith object occurrence present in the multiset m. For instance, given the multiset $m = aabcc$, then $m(4).list$ denotes the string of identifiers associated to the first occurrence of c, reading the string m from left to right.

Given a string of identifiers $list$, we denote by $list-$ the new string of identifiers obtained by removing the rightmost symbol from the string $list$; the addition of a symbol to a string of identifiers is done by using the standard string concatenation, indicated here by $+$ to avoid confusions. Given a string of identifiers $list$, then $last(list)$ denotes the rightmost symbol (then the rightmost identifier) of the string.

As the reader can see, the string of identifiers is actually used like a stack, and this is enough for our goal, due to the hierarchical structure of a membrane system. The other operations involving a string of identifiers will be not formally defined, since their meaning is rather intuitive. Moreover boolean predicates over the set of identifiers will also be used.

The definition of the model used here as follows.

Definition 1. *An evolution-communication P system with active membranes (in short,* an EC_{am} P system*), of degree $m \geq 1$, is defined as*

$$\Pi = (O, H, ID, \mu, w_1, w_2, \ldots, w_m, R),$$

where:

- *O is the alphabet of Π; its elements are called objects; to each occurrence of an object present in the system is associated a string (possibly empty) over the alphabet ID;*
- *H is a finite set of labels for membranes;*
- *ID is the set of identifiers; at any time, each membrane of the system has associated, in a unique way, one of the identifiers present in ID; as a general assumption we suppose that the set ID is finite but large enough to identify, at any time, all the membranes present in the system;*
- *μ is a membrane structure consisting of m membranes labelled with elements in H, not necessarily in a one-to-one manner;*

- *w_i, $1 \le i \le m$, specifies the multiset of objects present in the corresponding region i when the system is created; to each occurrence of an object is associated an empty string of identifiers; if an object has an unbounded number of occurrences, then we indicate it by using the index ∞;*
- *R is a set of developmental rules, of the following forms:*
  (a) : $[_h u \rightarrow v]_h$ or $[_{h_1}[_{h_2} u \rightarrow v]_{h_2}]_{h_1}$, for $h, h_1, h_2 \in H, u \in O^+, v \in O^+$, with $|u| \ge |v|$;
  ***object evolution** rules, associated with membranes and depending on the label of the membrane (or on the labels of the membranes in the second case) but not involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor modified by them; the objects present in the multiset u are rewritten into the objects described by the multiset v; each object occurrence in v is associated with a string of identifiers chosen as one of the strings of identifiers associated to the object occurrences present in u; formally, if $|u| = m$, and $|v| = n$, then we set $v(j).list = u(i).list$, for $1 \le i \le m, 1 \le j \le n$;*
  (b) : $(\mu_1 = [_{h_1} u[_{h_2} v]_{h_2}]_{h_1} \rightarrow \mu_2 = [_{h_1} v'[_{h_2} u']_{h_2}]_{h_1}, c)$ $h_1, h_2 \in H, u \in O^+, v \in O^+$;
  ***antiport** rules, that realize a synchronized exchange of objects; they are associated with membranes and depend on the labels of the membranes but do not change the membranes; the objects in u' and v' are exactly the objects in u and v, respectively (the objects are only moved), except for the associate strings of identifiers, updated in the following manner.*
  Suppose $|v'| = |v| = n$; then for $1 \le i \le n$, we set

$$v'(i).list = \begin{cases} v(i).list- & \text{if } last(v(i).list) = \mu_1(1).id, \\ v(i).list + \mu_1(2).id & \text{if } last(v(i).list) \neq \mu_1(1).id. \end{cases}$$

  The identifiers associated to the objects in u' are updated in a symmetric way. Suppose $|u'| = |u| = m$, then, for $1 \le i \le n$, we set

$$u'(i).list = \begin{cases} u(i).list- & \text{if } last(u(i).list) = \mu_1(2).id, \\ u(i).list + \mu_1(1).id & \text{if } last(u(i).list) \neq \mu_1(2).id. \end{cases}$$

  The idea that leads the update of the strings of identifiers is the following one. If an object enters exactly the membrane whose identifier is the last one inserted in the associated string of identifiers then the last identifier is removed (because the object is coming back in "its" region); if this is not the case, then the identifier of the membrane that the object is leaving, is added to the string of identifiers; c is a boolean predicate over the set of identifiers; if it is specified, then the rule is active, if and only if, the boolean predicate is true;
  (c) : $(\mu_1 = [_{h_1}[_{h_2} u]_{h_2}]_{h_1} \rightarrow \mu_2 = [_{h_1}[_{h_2}]_{h_2} u']_{h_1}, c)$
  $(\mu_1 = [_{h_1} u[_{h_2}]_{h_2}]_{h_1} \rightarrow \mu_2 = [_{h_1}[_{h_2} u']_{h_2}]_{h_1}, c)$, for $h_1, h_2 \in H, u \in O^+$;
  ***symport** rules, that move objects from inside to outside a membrane, or viceversa; they are associated to the membranes and depend on the labels but do not involve any change on the membranes; the objects in u' are exactly the same*

in u (the objects are only moved), except their associated strings of identifiers updated in the following way.
Suppose $|u'| = |u| = n$; then, for any $1 \leq i \leq n$, we set, in the first case,

$$u'(i).list = \begin{cases} u(i).list- & \text{if } last(u(i).list) = \mu_1(1).id, \\ u(i).list + \mu_1(2).id & \text{if } last(u(i).list) \neq \mu_1(1).id. \end{cases}$$

In the second case,

$$u'(i).list = \begin{cases} u(i).list- & \text{if } last(u(i).list) = \mu_1(2).id, \\ u(i).list + \mu_1(1).id & \text{if } last(u(i).list) \neq \mu_1(2).id. \end{cases}$$

The string of identifiers associated to the objects in u' are updated like in the case of antiport rules described above; if the boolean predicate c is specified, then the rule is active, if and only if, the specified predicate is true;
(d) : $[_i u]_i \rightarrow [_i v[_h w]_h]_i,\ h, i \in H, w, u, v \in O^+,\ |u| \geq |v| + |w|$;
***creation** rules; in reaction with some specified objects present in membrane i, a new membrane with label h is created inside the one with membrane i, and having inside the objects specified by w; the strings of identifiers, associated to the objects in w are the empty strings; to the new created membrane is associated a certain identifier from the set ID, not assigned to any membrane currently present in the membrane system; the objects present in u are possibly changed and the strings of identifiers for the objects in v are fixed like in the case of the object evolution rule $u \rightarrow v$;*
(e) : $[_{h_1} u]_{h_1} [_{h_2} u']_{h_2} \rightarrow [_{h_2} [_{h_1} v]_{h_1} v']_{h_2},\ |u| + |u'| \geq |v| + |v'|$;
***endocytosis**; an elementary membrane labelled h_1 enters the adjacent membrane labelled h_2 that can be non-elementary, under the control of the objects in u, u' and labels h_1, h_2; the labels of the two membranes remain unchanged; the objects in u and u' are possibly changed; the strings of identifiers of the objects in v and v' are fixed like in the case of the objects evolution rules, $u \rightarrow v$ and $u' \rightarrow v'$, respectively; all the remaining objects present in membrane h_1 and h_2 as well their strings of identifiers are left unchanged in the same membrane;*
(f) : $[_{h_2} [_{h_1} u]_{h_1} u']_{h_2} \rightarrow [_{h_1} v]_{h_1} [_{h_2} v']_{h_2},\ |u| + |u'| \geq |v| + |v'|$;
***exocytosis**; an elementary membrane labelled h_1 is sent out of a membrane labelled h_2, under the control of the objects in u and u' and the labels h_1, h_2; the labels of the two membranes remain unchanged; the objects in u and u' are possibly changed; the strings of identifiers of the objects in v and v' are fixed like in the case of objects evolution rules, $u \rightarrow v$ and $u' \rightarrow v'$, respectively; all the remaining objects present in membrane h_1 and h_2 as well their strings of identifiers are left unchanged in the same membrane.*

The evolution of the membrane system is defined in the following way. It starts from an initial configuration represented by the initial membrane structure μ and the multiset of objects available in its compartments at the beginning, hence $(\mu, w_1, w_2, \cdots, w_m)$. Some of the objects, in some of the regions, can be present in an unbounded number of occurrences. As mentioned, to each membrane is associated a unique identifier taken from the set of identifiers; such set is finite but we suppose is big enough to contain an unique identifier for each membrane present in the configuration of the system, during any possible computation; this assumption is justified by the fact that the number of membranes that can be created is bounded by the physical resources of the software system we are modelling.

To each object present in the regions of the membrane system is associated a string of identifiers (initially empty) updated during the evolution of the system in the way specified by the rules.

A single step of the evolution of the membrane system is composed by two sub-steps, applied one after the other one. In the first sub-step, the rules of type (a), (b), (c) are used in a non-deterministic maximally parallel manner.

The non-deterministic maximally parallel manner means that we assign the object occurrences to the rules, non-deterministically choosing the rules and the objects, until no further assignments is possible.

These rules realize the evolution of the objects by using the objects evolution rules and the movement of the objects by using the symport/antiport rules.

Each object occurrence can be used by only one of these rules. The strings of identifiers associated to the objects involved are updated as specified by the rules. The symport/antiport rules are active only for the membranes where the boolean predicate, if any, specified by c, is true.

In the second sub-step, the rules of type (d) - (f), are used, still in a maximally parallel non-deterministic way. These rules modify the membrane structure of the membrane system. Each object occurrence and each membrane can be involved in only one of these rules.

Also in this case, the strings of identifiers associated to the involved objects are updated in the way specified by the rules.

Notice that never during the entire evolution new object occurrences are created; they are only transformed.

The evolution of the membrane system halts when the system reaches a halting configuration, that is a configuration where no rules can be further applied. On the other hand, there might be membrane systems whose evolution can run forever.

3 Membrane Systems as Dynamic Software Architectures Specifications

In this section, first we show how membrane systems can be used to formally specify static architectures: the number and type of components and connections do not change. Successively, we address the problem of capturing dynamic architectures: systems for which composition of interacting components changes during the course of a single evolution.

A software architecture specification describes the high-level design of a system in terms of the structure of the components and their communication relationships.

Thus, the specification focuses on the interaction behaviors exhibited by the components, not on the algorithms used internally by the components to carry out their functional roles.

The idea of the proposed approach is to describe the components (or entities) of the system, which can be classes, procedures, processes, etc., with the membranes of the membrane system, which evolve and communicate according to the rules associated to them. In particular, the membrane systems description of a static software architecture consists of the mentioned hierarchical structure of membranes where:

1. the associated multisets of objects (abstract data) and the associated set of evolution rules (abstract manipulation of data) represent the static components of the architecture,
2. the associated sets of symport/antiport rules represent the connectors of the architecture.

For instance, consider the simple client-server system shown in Fig. 1(a). It consists of one client and one server interacting via a link. This system is described by the membrane system graphically depicted in Fig. 1(b). In particular, to each entity of the system corresponds a membrane, and the communication between these membranes is accomplished by antiport rules of the type $[_{server}\ r_2\ [_{client}\ r_1\ ]_{client}\ ]_{server} \rightarrow [_{server}\ r_1\ [_{client}\ r_2\ ]_{client}\ ]_{server}$ (shown in Fig. 1(b) with an arrow), which describe the request r_1 sent by the client to the server, and the response r_2 sent by the server to the client. The client membrane has also associated a multiset of objects representing its internal state, and a set of evolution rules that evolve the objects according to the requests made to the server and the responses received. As an example, when the previous antiport rule is applied, the object r_2 will activate some rules in the client membrane that evolve its internal state. Moreover, some of the evolution rules can be executed only if the client membrane is inside the server one, that is the rules are of the form $[_{server}\ [_{client}\ a \rightarrow b\ ]_{client}\ ]_{server}$.

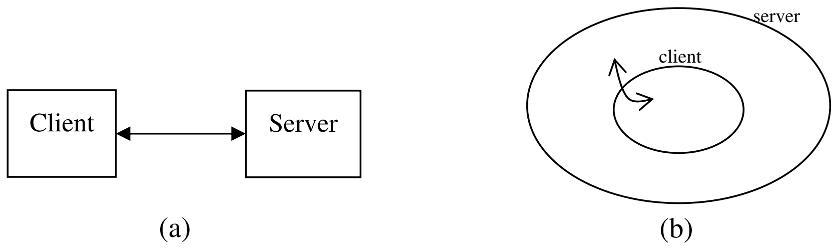

Fig. 1. A simple client-server system (a) and the membrane system modelling it (b).

Now, let us consider a client-server system that changes the configuration during the evolution. As an example, suppose that the system has two servers: a primary server that provides many services, but it may go down suddenly, and a secondary server that provides less services, but it is reliable. Thus, a client interacts with a primary server and if, and only if, the primary server fails then it would start to interact with the secondary one, as shown in Fig. 2(a) and 2(b).

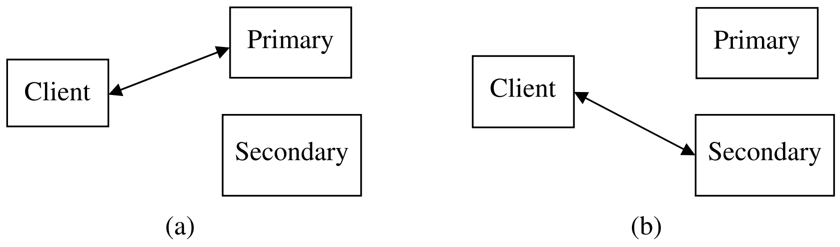

Fig. 2. A dynamic client-server system.

These dynamic aspects are supported by the membrane systems through the active membrane rules, which represent reconfiguration of the membrane structure. The behavior of the previous client-server system can be described by the membrane system graphically depicted in Fig. 3. Here new membranes representing the clients can be created in the environment by using creation rules, and can enter into the primary server membrane by using the endocytosis rule $[_{client}\ ]_{client}\ [_{primary} u]_{primary} \rightarrow [_{primary} u[_{client}\ ]_{client}]_{primary}$ if and only if the state of the primary server is up (i.e., it contains the object u) as in Fig. 3(a). In the case the primary server goes down, its state changes from u to d and the secondary server is activated by setting its state to up. Moreover, the client membrane leaves the primary server by using the exocytosis rule $[_{primary}\ d\ [_{client}\ ]_{client}]_{primary} \rightarrow [_{client}\ ]_{client}\ [_{primary}\ d\ ]_{primary}$, and enters into the secondary server by executing the endocytosis rule $[_{client}\ ]_{client}\ [_{secondary}\ u\ ]_{secondary} \rightarrow [_{secondary}\ u\ [_{client}\ ]_{client}]_{secondary}$, as shown in Fig. 3(b).

The exclusive access to the two servers for the clients is guaranteed by the following endocytosis and exocytosis rules:

$$[_{primary}\ u\ ]_{primary}\ [_{secondary}\ u\ ]_{secondary} \rightarrow [_{primary}\ [_{secondary}\ d\ ]_{secondary}\ u\ ]_{primary}$$
$$[_{primary}\ [_{secondary}\ d\ ]_{secondary}\ u\ ]_{primary} \rightarrow [_{primary}\ u\ ]_{primary}\ [_{secondary}\ d\ ]_{secondary}$$
$$[_{primary}\ [_{secondary}\ d\ ]_{secondary}\ d\ ]_{primary} \rightarrow [_{primary}\ d\ ]_{primary}\ [_{secondary}\ u\ ]_{secondary}$$
$$[_{primary}\ d\ ]_{primary}\ [_{secondary}\ d\ ]_{secondary} \rightarrow [_{primary}\ [_{secondary}\ u\ ]_{secondary}\ d\ ]_{primary}$$
$$[_{primary}\ [_{secondary}\ u\ ]_{secondary}\ d\ ]_{primary} \rightarrow [_{primary}\ d\ ]_{primary}\ [_{secondary}\ u\ ]_{secondary}$$
$$[_{primary}\ [_{secondary}\ u\ ]_{secondary}\ u\ ]_{primary} \rightarrow [_{primary}\ u\ ]_{primary}\ [_{secondary}\ d\ ]_{secondary}$$

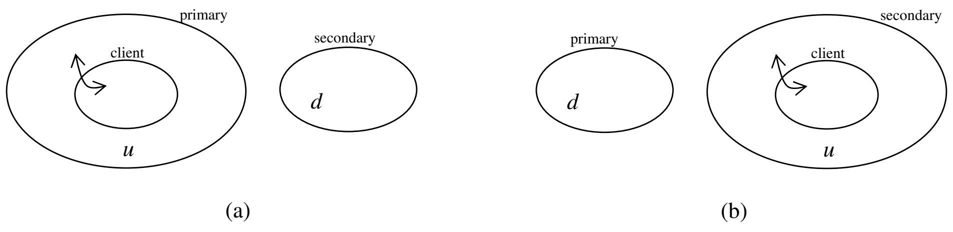

Fig. 3. The possible configurations of a dynamic client-server system.

An important information embedded into the rules of the membrane system is the style of the specified architecture, that is the possible interconnections between its individual components at run time. In fact, two membranes can be considered interconnected when the symport/antiport rules can be applied on them, this means that one of the two membranes is inside the other. These hierarchies are created by applying the creation and endocytosis rules present in the membrane system. Whereas, the dissolving and exocytosis rules disconnect the components of the architecture at run time.

4 Describing the Architecture of a System for Monitoring a Hospital Ward

In this section we show an example of membrane system for modelling the architecture of a software system monitoring a hospital ward. In this simple example we consider an architecture formed by the low-level components of the software system such as doctor, nurse, and patient. The dynamic behavior of the system is the following. When a patient is registered into the monitoring system its state of health is checked by the doctor and then it is assigned to a nurse. Each nurse can have in charge a fixed number of patients. Periodically the doctor checks the health of the patients and give them a dose of medicine. When a patient is restored, s/he leaves the nurse and its health state is periodically checked by the doctor.

The following EC_{am} P system represents a specification of the architecture for the monitoring system:

$$\Pi = (O, H, ID, \mu, w_{extenv}, w_{doctor}, R),$$

where:

$$\begin{aligned} O = & \ \{p, n, q, t\} \cup \{s_d \mid d \in \{1, \ldots, m\}\} \cup \{x_b, x'_b, x''_b \mid b \in \{1, \ldots, l\}\} \\ \cup & \ \{m_a \mid a \in \{1, \ldots, k\}\}; \\ H = & \ \{extenv, doctor, nurse, patient\}; \\ \mu = & \ [_{extenv} \ [_{doctor} \ ]_{doctor} \ ]_{extenv}; \end{aligned}$$

$$
\begin{aligned}
&\quad w_{extenv} = p^{\infty} n^{\infty};\\
&\quad w_{doctor} = \lambda;\\
R = \{&[_{extenv}\ p \rightarrow p\ ]_{extenv},\\
&[_{extenv}\ n \rightarrow n\ ]_{extenv},\\
&[_{extenv}\ pp]_{extenv} \rightarrow [_{extenv}[_{patient}\ s_d s_d\ ]_{patient}]_{extenv},\\
&[_{extenv}\ nnnnn\ ]_{extenv} \rightarrow [_{extenv}[_{nurse}\ ttttt\ ]_{nurse}]_{extenv},\\
&[_{patient}\ s_d s_d\ ]_{patient}[_{doctor}\ ]_{doctor} \rightarrow [_{doctor}[_{patient}\ x_b x_b\ ]_{patient}]_{doctor},\\
&[_{doctor}[_{patient}\ x_b\ ]_{patient}]_{doctor} \rightarrow [_{patient}\ x_b\ ]_{patient}[_{doctor}\ ]_{doctor},\\
&[_{extenv}[_{patient}\ x_b \rightarrow x_b\ ]_{patient}\ ]_{extenv},\\
&[_{extenv}[_{nurse}\ ttttt\ ]_{nurse}]_{extenv} \rightarrow [_{nurse}\ ttttt\ ]_{nurse}[_{extenv}\ ]_{extenv},\\
&[_{nurse}\ [_{patient}\ x'_b\ ]_{patient}]_{nurse} \rightarrow [_{nurse}\ x'_b[_{patient}\ ]_{patient}]_{nurse},\\
&[_{extenv}\ x'_b[_{doctor}\ ]_{doctor}]_{extenv} \rightarrow [_{extenv}\ [_{doctor}\ x'_b\ ]_{doctor}]_{extenv},\\
&[_{extenv}\ [_{doctor}\ m_a\ ]_{doctor}]_{extenv} \rightarrow [_{extenv}\ m_a[_{doctor}\ ]_{doctor}]_{extenv},\\
&(\mu_1 = [_{extenv}\ m_a[_{nurse}\ ]_{nurse}]_{extenv} \rightarrow \mu_2 = [_{extenv}\ [_{nurse}\ m_a\ ]_{nurse}]_{extenv},\\
&last(m_a.list) = \mu_1(2).id),\\
&(\mu_1 = [_{nurse}\ m_a[_{patient}\ ]_{patient}]_{nurse} \rightarrow \mu_2 = [_{nurse}\ [_{patient}\ m_a\ ]_{patient}]_{nurse},\\
&last(m_a.list) = \mu_1(2).id),\\
&\mid d \in \{1,\dots,m\}, a \in \{1,\dots,k\}, b,c \in \{1,\dots,l\},\\
&x_b \text{ is the health state associated to symptom } s_d\}\\
&\cup R_t \cup R_d \cup R_p;\\
R_t = \{&[_{patient}\ x_b\ ]_{patient}[_{nurse}\ t\ ]_{nurse} \rightarrow [_{nurse}[_{patient}\ x_b\ ]_{patient}]_{nurse}\\
&\mid b \in \{1,\dots,l\}, x_b \text{ indicates an ill health state}\}\\
&\cup\{[_{patient}\ x_b x_b\ ]_{patient}[_{doctor}\ ]_{doctor} \rightarrow [_{doctor}[_{patient}\ x_c x_c\ ]_{patient}]_{doctor}\\
&\mid b,c \in \{1,\dots,l\}, x_b \text{ indicates a restored health state}\};\\
R_d = \{&[_{doctor}\ x'_b \rightarrow\ m_a\ ]_{doctor} \mid m_a.list = x'_b.list, a \in \{1,\dots,k\}, b \in \{1,\dots,l\},\\
&m_a \text{ is the medicine associated to the health state } x'_b\};\\
R_p = \{&[_{nurse}[_{patient}\ x_b x_b \rightarrow x'_b x''_b]_{patient}]_{nurse},\ [_{patient}\ m_a x''_b \rightarrow x_c x_c]_{patient}\\
&a \in \{1,\dots,k\}, b,c \in \{1,\dots,l\},\\
&m_a \text{ is the medicine that transforms the health state } x''_b\\
&\ \text{into the health state } x_c\}\\
&\cup\{[_{nurse}[_{patient}\ x_b\ ]_{patient}]_{nurse} \rightarrow [_{patient}\ x_b\ ]_{patient}[_{nurse}\ t\ ]_{nurse}\\
&\mid b \in \{1,\dots,k\}, x_b \text{ indicates a restored health state}\}.
\end{aligned}
$$

The P system has four "types" of membrane, *extenv* corresponds to the environment outside the monitoring system, the other labels correspond to *doctor*, *nurse* and *patient* entities. In the initial configuration, the P system is formed by an external environment membrane embedding a doctor membrane. The objects

p and n active the rules for creating a patient and a nurse membrane, respectively. The object s_d is associated to the patient membranes and represents the symptoms of the patient, their total number is m. Each object t associated to a nurse membrane indicates a patient that can taken in charge. In this example, each nurse can have at most five patients in charge since five t are initially associated to each nurse membrane. The objects x_b indicates the health states of the patients, whereas the objects m_a are the medicines prescribed by the doctor. Their total number is l and k, respectively. The set of health states can be partitioned in the set of *restored health states* and the set of *ill health states*.

The constructed P system simulates the dynamic aspects of the monitoring system in the following way. Starting form the initial configuration, in region *extenv* there are an unbounded number of occurrences of objects p and n, which can be rewritten by the first and second rule in R or can be used to create patient and nurse membranes by using the third and fourth rule in R; the rule selection is done in a non deterministic way. The creation of the membranes simulates the registration of new patients and nurses to the monitoring system. The patient has associated two objects s_d indicating the symptoms.

The patient moves into the doctor membrane by using the fifth rule, which substitutes the symptom s_d of the patient with two occurrences of x_b indicating the health state. Successively, the patient leaves the doctor membrane by using the sixth rule in R, which is an exocytosis rule. Now, the patient can be assigned to an available nurse (i.e., that has a t in its membrane) by using the first rule in R_t if its x_b is an ill health state. When the health state of a patient is restored, the corresponding membrane leaves the nurse by using the last rule in R_p. The eighth rule of R allows a nurse membrane to leave the monitoring system if it has not patient in charge.

Let us now focus our attention on the evolution of a certain patient membrane. Once the doctor has assigned an ill health state x_b to the patient, the patient is introduced in a nurse membrane and the two occurrences of x_b are changed into x'_b and x''_b by using the first rule in R_p. Then x'_b is sent to the doctor membrane by using the ninth and tenth rule in R. Once in the doctor membrane, the health state x'_b is changed into a medicine m_a that is one of the medicine associated to such health state.

Notice that the object corresponding to the medicine has associated the same string of identifiers associated to the health state, indicating in this way the path that the medicine has to do in order to reaches the correct patient. The last rules in R use the identifiers associated to m_a for moving the medicine to the patient. Finally, by using the second rule in R_p the health state of the patient evolves according to the medicine prescribed by the doctor. This process is repeated for each patient until its health state is restored. In this case, the patient leaves the nurse membrane and its health state is periodically checked by the doctor by using the last rule in R_t.

Figure 4 shows a configuration of the P system. In particular, the external environment contains a recovered patient (its current health state is indicated by the occurrence x_1), a patient just arrived (contains the symptom s_2s_2), a patient that has been checked by the doctor (its health state is x_5), two nurses having in charge one and two patients, respectively. The doctor has just received the health state (x'_2) of the patient having the object x''_2, whereas the patient with health state x''_4 has just requested a medicine. At the next step the patient with health state x''_7 will receive the medicine m_9 prescribed by the doctor, and its health state will evolve with the rule $m_9x''_7 \rightarrow x_c$, with x_c a new health state.

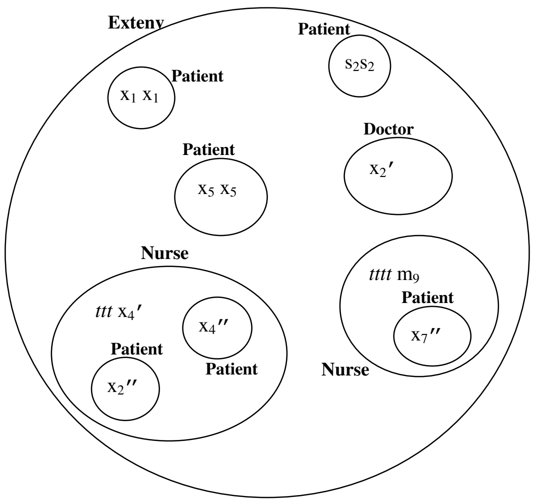

Fig. 4. A configuration of the EC_{am} P system modelling the architecture of a system for monitoring a hospital ward.

The system can be easily extended to support multiple wards. As an example, Fig. 5 shows a P system configuration describing the architecture of a monitoring system with several wards. In this system, when a patient membrane enters into the system, a doctor determines to which ward the patient must be assigned. Each ward contains a head physician that now works like the doctor in the above described P system.

5 Studying the Behavior of the System: Some Considerations

As mentioned in the Introduction one of the goal of the formal modelling of a software system is to study properties of the designed system in a formal way.

A possible way to study properties of the system is to analyze the behavior of the system, meaning with that, the sequence of configurations touched by the systems during its evolutions. The study of the behavior of a membrane system by using an appropriate observer has been introduced in [5].

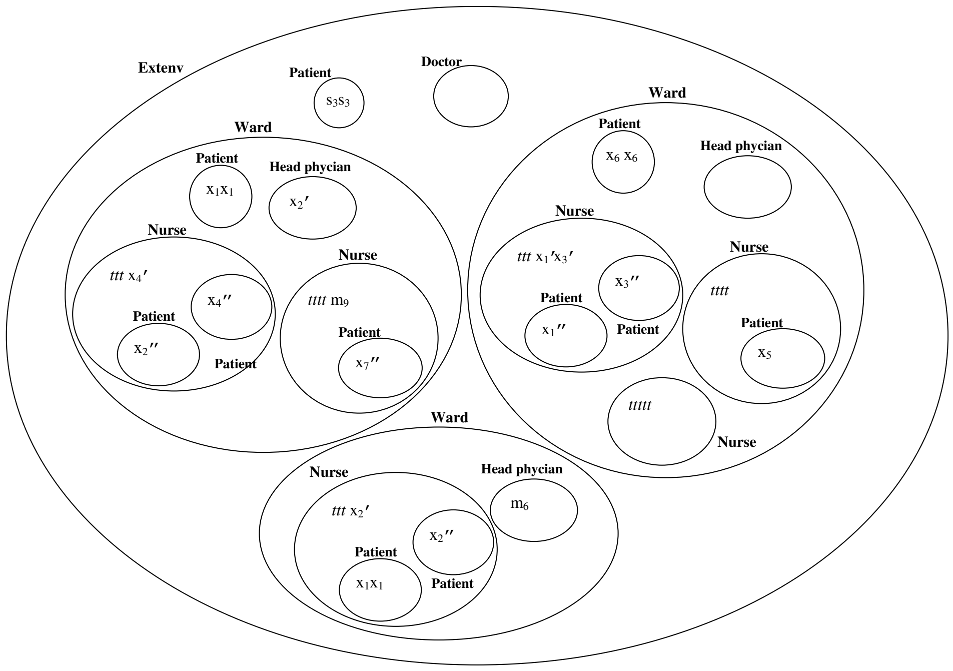

Fig. 5. A configuration of a EC_{am} P system modelling the architecture of a system for monitoring a hospital with multiple wards.

It is rather natural to consider the behavior of a membrane system like a string of symbols, describing the sequence of configurations touched during the evolution of the system. This approach can be used only when is possible to associate to each configuration of the system a unique symbol of a finite alphabet; so it cannot be used when the number of distinct configurations of the system is infinite – and this is the case also for the P systems defined in the definition given in Section 2.

In fact, the presence of objects in unbounded number of occurrences of objects in the initial configuration and the possibility to create new membranes guarantees that the possible number of configurations reachable by the systems, during the evolution, is infinite. On the other hand this fact guarantees that the model defined in Section 2 is computationally universal (since it joins features, like antiports, cooperative rules, of well-known universal models, [14]) and so, in informal words, it can be used to model "everything". Anyway, this computational universality is a problem from a practical point of view; it is well-known that in a universal model all non-trivial properties are undecidable, [7].

A possible solution to this problem is to model a system by using a (universal) complete model (like the one defined here) but then to observe, in a clever way, only a (sub)part of the model, studying properties only on the observed part.

The idea is to get a clever observation that, on one hand, still preserve the "information" of the system, and, on the other hand, make easier to decide formal properties.

This approach is similar to what has been done in [5] where a formal observer has been introduced to study the behavior of a P system.

First, we fix some necessary notation. Σ denotes a finite non-empty alphabet, Σ^* the free monoid generated by this alphabet with concatenation, and λ, the empty word, as its neutral element. When we speak about infinite words, w^ω denotes the infinite catenation of w to the right and this is the only type of infinite words we will consider. Thus Σ^ω is the set of all infinite words, and $\Sigma^\infty := \Sigma^* \cup \Sigma^\omega$ is the set of all words.

For other details about the basic knowledge of formal languages, finite, infinite words and Büchi automata we suggest the reader to consult the books [2, 7].

Given a P system Π, each configuration can be represented as a string of brackets, representing the hierarchical structure of the membrane, together with their associated contents.

Two strings represent the same configuration if one can be obtained by the other one by exchanging pairs of brackets (together with their contents) representing regions at the same level in the system.

So, given a configuration c there are several strings representing c. All these strings can be collected in a set and we choose a unique string in the set, representing the entire set. For simplicity of notation, when we refer to a configuration c then c would formally denotes the string representing the set of all strings representing the configuration c.

A P system Π, moves, at each *step*, from a configuration to the next one, by using specified rules.

We define:

$$C_\Pi = \{c \mid c \text{ is a configuration that the system } \Pi \text{ reaches during a computation}\},$$

that is, the set of all possible configurations that the system Π can reach, during any computation. Notice that the set C_Π can be finite or infinite, depending on the system Π considered.

We construct a finite partition *obs*, standing for *observer*, of the set C_Π. Let $obs = C_{\Pi_1} \cup C_{\Pi_2} \cdots \cup C_{\Pi_k}$. Given a configuration c, then $[c]_{obs}$ indicates the (unique) set of the partition *obs* where the configuration c is present. We can construct a labelling function l_{obs} that associates to each set $C_{\Pi_i}, 1 \leq i \leq k$, a unique label taken from a finite set of labels Σ.

For a sequence *seq* of configurations of Π, $seq = c_0, c_1, \cdots, c_k, \cdots$, we shortly write $l_{obs}(seq)$ for the string $l_{obs}([c_0]_{obs}) \cdot l_{obs}([c_1]_{obs}) \cdots l_{obs}([c_k]_{obs}) \cdots$.

We now can define the *behavior* $B_{obs}(\Pi)$ of the system Π, according to the partition *obs*:

$$B_{obs}(\Pi) = \{l_{obs}(seq) \mid seq \in SEQ_\Pi\},$$

where SEQ_{Π} is the set of all possible sequences of configurations during any computation (halting or non halting) of the system Π.

Notice that $B_{obs}(\Pi)$ is a language consisting of finite and infinite words over the alphabet Σ. Simply by intersecting $B_{obs}(\Pi)$ with a regular language we can select *finite behavior* $B^{fin}_{obs}(\Pi)$ and *infinite behavior* $B^{inf}_{obs}(\Pi)$.

Formally:

$$B^{fin}_{obs}(\Pi) = B_{obs}(\Pi) \cap \Sigma^*,$$

$$B^{inf}_{obs}(\Pi) = B_{obs}(\Pi) \cap \Sigma^{\omega}.$$

The partition *obs* plays the role of an observer of the system Π: it groups, in a specified way, the configurations of the system Π.

The defined languages may be too complex to be investigated. In fact, it is known, that, for every recursively enumerable language L is possible to construct a (simple) system Π, a partition *obs* such that $B^{fin}_{obs}(\Pi) = L$ (for details we refer to [5]).

Therefore, it seems necessary to find two languages, $\widehat{B}^{obs}_{fin}(\Pi)$ and $\widehat{B}^{obs}_{inf}(\Pi)$ that are "approximations" of the behaviors $B^{fin}_{obs}(\Pi)$ and $B^{inf}_{obs}(\Pi)$. The definition of approximation is not formally given here and is left for the future work. For our goal here is enough to have the intuitive idea that, given a language L, then the language $\widehat{L}$ approximates L if $L \subseteq \widehat{L}$. Of course, a rather trivial approximation of any language L over an alphabet Σ is the language Σ^*. But intuitively this approximation is "too far" from L because it loses every information on the original language.

Therefore the problem is to find good approximations, meaning with that a language easy to analyze but not too far from the original language.

Coming back to our goal, given a behavior B (infinite or finite), we need to find a behavior $\widehat{B}$ that is an approximation of B.

A possible approximation of a behavior is called *automata approximation* and it is constructed in the following way.

Given a P system Π and a partition $obs = C_{\Pi_1} \cup C_{\Pi_2} \cdots \cup C_{\Pi_k}$ of the set C_{Π}, we construct a finite state automaton $A' = (K, V, s_0, \delta, F)$ in the following way. Each state in K corresponds in a unique way to one of the subset composing *obs*. The input alphabet V coincides with the set of states. The initial state s_0 corresponds to the subset in *obs* having the initial configuration of Π. The set f of states corresponds to the subsets of *obs* containing a halting configuration. The transition δ is fixed in the following way. $\delta(C_{\Pi_i}, C_{\Pi_j}) = C_{\Pi_j}$, $1 \leq i, j \leq k$, if and only if, the system Π can move, in one step of evolution, from a configuration present in C_{Π_i} to a configuration present in C_{Π_j}.

In similar way we can construct a Büchi automaton A'', except that in this case the set of final states F coincides with K. It is clear that $B^{fin}_{obs}(\Pi) \subseteq L(A')$ and $B^{inf}_{obs}(\Pi) \subseteq L(A'')$ (so $L(A')$ and $L(A'')$ are approximations of $B^{fin}_{obs}(\Pi)$ and $B^{inf}_{obs}(\Pi)$, respectively).

Notice that the procedure given may be not effective. For this reason we say that a behavior can be *automata approximated* if and only if the automaton defined earlier can be effectively constructed.

Moreover, a system Π can be automata approximated if, and only if, its behavior can be automata approximated.

If the system Π has a finite number of configurations (i.e., the cardinality of C_Π is finite), then the system can be automata approximated by using the trivial partition obs_{total} constructed by just considering each configuration in C_Π as a unitary set of the partition obs_{total}. Actually, in this trivial case we do not have an approximation but actually the equality $B^{fin}_{obs_{total}}(\Pi) = L(A')$ and $B^{inf}_{obs_{total}}(\Pi) = L(A'')$.

Find good approximations for complicate behaviors is a rather challenging topic, with several possible implications in different fields.

On the other hand, it is also interesting to individuate classes of systems that can be automata approximated when using specified observers. These two problems are left as open problems for further investigations.

We conclude by showing that it is possible to construct an observer *obs* such that the system defined in Section 4 can be automata approximated and is possible to investigate properties of the system by using such an approximation.

Informally, to construct the observer of the system, one needs to fix the components of the system that have to be investigated. For instance, in our case, one of the components of the system that may be interesting to study is the *patient.*

Initially we individuate the states of the patient which we consider interesting for our investigation. For instance, these might correspond to: *patient with some symptoms, recovered in the hospital but not yet visited by a doctor*; *patient visited by the doctor*; *patient under treatment*; *patient that is taking medicine*; *patient restored.*

Now, suppose that in the original configuration of the system Π, there exists a patient represented by a membrane with label *patient* and with identifier *id.*

Then, it is possible to partition the set of all reachable configurations C_Π according to the states of such specified patient.

In particular, we can construct the partition of C_Π, $obs_{patient} = C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$, where:

$C_1 = \{c \mid$ the membrane with identifier id contains the occurrence s_d, $d \in \{1, \cdots, m\}\}$.

This set contains all the configurations in C_Π where the specified patient has certain symptoms (indicated by s) but has not yet been visited by the doctor.

$C_2 = \{c \mid$ the membrane with identifier id contains two occurrences of x_b and the membrane is inside a membrane with label *doctor*, $b \in \{1, \cdots, l\}\}$.

This set contains all the configurations in C_Π where the specified patient has been visited by the doctor; x_b indicates the health state associated by the doctor to the patient, according to the symptoms.

$C_3 = \{c \mid$ the membrane with identifier *id* contains two occurrences of x_b, the membrane is inside a membrane with label *nurse* and x_b is not a restored health state, $b \in \{1, \cdots, l\}\}$.

This set contains all the configurations in C_Π where the patient is under treatment (it is associated to a nurse); x_b indicates the current health state of the patient.

$C_4 = \{c \mid$ the membrane with identifier *id* contains an occurrence of $m_a, a \in \{1, \cdots k\}\}$.

This set contains all the configurations where the patient is taking a medicine; m_a identifies the medicine.

$C_5 = \{c \mid$ the membrane with identifier *id* contains an occurrence of x_b and x_b is a restored health state, $b \in \{1, \cdots, l\}\}$.

This set contains all the configurations in C_Π where the patient has been restored; x_b indicates a restored health state. It is possible to see that $obs_{patient}$ is a partition of C_Π.

We are interested in $B^{inf}_{obs_{patient}}(\Pi)$. Looking at the P system Π described in Section 4 we can construct the finite state automata that approximate $B^{inf}_{obs_{patient}}(\Pi)$, in the way previously described.

In particular, looking to the rules of Π and following the procedure given earlier is possible to construct the Büchi automaton represented in Fig. 6.

The language accepted by this automaton is exactly $B^{inf}_{obs_{patient}}(\Pi)$.

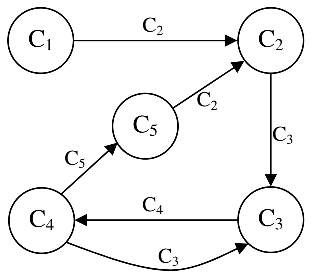

Fig. 6. The automaton for $B^{inf}_{obs_{patient}}(\Pi)$.

The automaton can be used to prove properties of the constructed system Π. For instance, using the automaton it is possible to infer two important properties of the system: the patient cannot take a medicine if s/he is not under treatment because all the paths from state C_1 to state C_4 pass through state C_3; moreover, is also true that a patient cannot be under treatment if it has not been visited by a doctor (in fact, the state C_3 in the automaton can be reached only passing through the state C_2).

We can also see that the patient evolves in a deterministic way, that means given a state (among the ones investigated), there is an unique possible next state (among the ones investigated), except for the state C_4; in this case, in fact, the patient can either be restored (state C_5) or can remain under treatment (state C_3).

It is possible to construct several automata, for each possible component we are interested. For instance, it is possible to investigate the behavior of a nurse and of a doctor by using the same approach used for a patient. Moreover it is also possible to investigate properties concerning the entire system and the interactions between the different components by using the cross-product operation, [17], among the realized automata. On the other hand, the entire problem of approximation can be formally investigated in the framework of rough sets theory dedicated to the formal study of approximation, [16]. We leave these interesting topics for further works.

References

1. R. Allen, D. Garlan: A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6, 3 (1997), 213–249.
2. J. Berstel, J.E. Pin: *Infinite Words.* Elsevier, Amsterdam, 2004.
3. N. Carriero, D. Gelerntern: Linda in context. *Communications of the ACM*, 32, 4 (1989), 444–458.
4. M. Cavaliere: Evolution–communication P systems. In *Membrane Computing* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), LNCS 2597, Springer-Verlag, Berlin, 2003, 134–145.
5. M. Cavaliere, P. Leupold: Evolution and observation – A new way to look at membrane systems. In *Membrane Computing* (C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2933, Springer-Verlag, Berlin, 2004, 70–88.
6. A.A. Holzbacher: A software environment for concurrent coordinated programming. *Proc. First International Conference Coordination Models, Languages and Applications*, LNCS 1061, Springer-Verlag, Berlin, 1996, 249–266.
7. J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
8. P. Inverardi, A.L. Wolf: Formal specification and analysis of software architecture using the chemical abstract machine model, *IEEE Transactions On Software Engineering*, 21, 4 (1995), 373–386.
9. J. Kramer: Configuration programming. A framework for the development of distributable systems. *Proc. COMPEURO '90, IEEE*, 1990, 374–384.
10. N. Medvidovic, R.N. Taylor: A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering*, 26, 1 (2000), 70–93.
11. D. Le Métayer: Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24, 7 (1998), 521–533.
12. Gh. Păun: Computing with membranes: Attacking NP-complete problems. In *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen eds.), Springer-Verlag, London, 2000, 94–115.
13. Gh. Păun: P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6, 1 (2001), 75–90.
14. Gh. Păun: *Membrane Computing. An Introduction.* Springer-Verlag, Berlin, 2002.

15. Gh. Păun, G. Rozenberg: A guide To membrane computing. *Theoretical Computer Science*, 287, 1 (2002), 73–100.
16. Z. Pawlak: Rough sets. *International Journal of Computer and Information Sciences*, 11 (1981), 341–356.
17. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages.* Springer-Verlag, Berlin, 1997.
18. D. Soni, R.L. Nord, C. Hofmeister: Software architecture in industrial applications. *Proc. of the Int. Conf. on Software Engineering*, 1995, 196–207.
19. C. Szyperski: Component technology: What, where, and how? *Proc. of the Int. Conf. on Software Engineering*, 2003, 684–693.
20. M. Wermelinger: Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145, 5 (1998), 130–136.
21. The P Systems Web Page: http://psystems.disco.unimib.it