

Formal Verification and Testing Based on P Systems

Marian Gheorghe^{1,2}, Florentin Ipatе², and Ciprian Dragomir¹

¹ Department of Computer Science, The University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK

`M.Gheorghe@dcs.shef.ac.uk`

² Department of Computer Science
Faculty of Mathematics and Computer Science

The University of Pitești
Str. Târgu din Vale 1, 110040 Pitești
`florentin.ipate@ifsoft.ro`

Abstract. In this paper it is surveyed the set of formal verification methods and testing approaches used so far for applications based on P systems.

1 Introduction

P systems (also called membrane systems) represent a class of parallel and distributed computing devices which are inspired by the structure and functioning of the living cells [18], [19]. The model has been intensively investigated from a theoretical perspective, and studies related to computational power, complexity aspects, hierarchies of different mechanisms and connections with other similar models have been undertaken. Another important line of research has considered P systems as a vehicle to represent different problems from various domains [20]. A rich set of software tools has been produced, implementing simulators for various classes of P systems or translators to formal verification tools [11].

As a consequence of using membrane systems to specify, model and simulate various systems, certain methods and techniques have been employed to verify they produce the expected results.

Formal methods have been used for various types of membrane systems and using different formalisms. Petri nets have been used to express the semantics of certain classes of P systems and methods to translate these systems into Petri nets have been developed. Tools and techniques produced and utilised for Petri nets become available for the description, analysis, and verification of membrane systems [17]. They also allow to study specific properties of such systems, like causality and (a)synchrony.

Structural operational semantic for certain some classes of P systems has been systematically investigated and their translation into specific rewriting logic formalisms provided by Maude [9], [5], has been defined. This approach allows to formally verify properties of such systems by using linear temporal logic model checking procedures [3].

For probabilistic and stochastic P systems special relationships with classes of stochastic process algebras and Petri nets have been investigated and a special purpose model checking approach based on PRISM, which will be discussed in the next section, has been studied [6].

A complementary approach to formal verification is usually based on testing. In the case of P systems this route has been recently open to research by considering some classical test coverage criteria [14]. More specifically, model based testing has been investigated for simple classes of P systems [14], [16] and ways to devise adequate test sets have been proposed. These techniques are somehow similar to studies investigating the role of the so called observers [7], [8] for certain classes of P systems where the behaviour is just filtered through some mechanisms for a well-defined purpose. In the case of testing the behaviour which is selected needs to obey some testing rules. The problem of testing P systems will be surveyed in section 3.

2 P Systems Verification

In many research areas, like, cellular biology, ecology, social insects study, the accurate simulations play a very important role as they reveal new properties that can be difficult or impossible to discover through direct experiments. One key question is what one can do with a model, other than just simulate trajectories. The problem of formal verification of the behaviour of such systems represents an alternative to simulations for verifying certain properties and for revealing new behaviour. Formal verification methods have been studied for various classes of P systems. This research can be classified according to certain criteria that reveal various formal aspects of such systems.

Due to a rapid proliferation of various variants of P systems, the research on various formal semantics started quite early and consists of a wealth of different approaches. Structural operational semantics for basic classes of P systems [3] together with rewriting logic semantics [5] offer the appropriate framework to develop not only rigorous formal definitions of certain classes of P systems, but also the opportunity to define adequate translations of such systems into a well-known model checker, Maude [5]. Using this framework a large variety of queries can be formulated in Linear Temporal Logic (LTL for short) and then verified using suitable Maude tools [3]. Executable semantics based on an extension to Maude, called K, is provided for certain classes of P systems [22]. Another class of semantics is based on Petri nets [17] and it brings also the wealth of tool support that comes together with these models. Finally, semantics aiming to include compositionality aspects using a process algebra style, is studied for very basic classes of P systems [4].

There have been studies on mapping various models of P systems into other computational models that benefit from well-established formal verification and testing methods. A class of bounded symport/antiport is coded as brane calculi [24] and basic membrane systems are matched against X-machines [1].

Relationships between membrane systems and other formalisms, like, ambients [2], cellular automata [13], Petri nets [12], and X-machines [23], have been

also considered. Specific model-checking decidability results for P systems have been studied [10]. More precisely, decidability results for queries formulated in Computation Tree Logic (CTL for short) augmented with atomic predicates in REG and LIN, are investigated for the class of bounded P systems – only rewriting rules with the left hand side bigger than the right hand side are utilised.

The problems described above have been considered for various classes of deterministic or nondeterministic P systems. Stochastic systems behave in an unintuitive way and consequently are harder to conceive and verify. The field is widely open for theoretical investigations and applications in various areas. A way of formally verifying stochastic behaviour is to use specific model checking tools to analyse in an automatic way various properties of the model. The class of P systems which is suitable for such formal verification consists of stochastic P systems.

In most of the variants of P systems the rules are applied in a maximally parallel way. This mechanism allows an efficient execution of the systems and proves to be very effective for theoretical investigations where it plays a major role in many circumstances. However in applications where the number of molecules is not that big, a different variant seems to be more suitable. This variant has been introduced in [21] and will be discussed in some detail below.

Definition 1. *A stochastic P system is a construct*

$$\Pi = (O, L, \mu, M_1, M_2, \dots, M_n, R_1, \dots, R_n)$$

where:

- O is a finite alphabet of symbols, called objects;
- L is a finite alphabet of labels associated with compartments;
- μ is a membrane structure containing $n \geq 1$ membranes labelled by elements from L ;
- $M_i = (l_i, w_i, s_i)$, for each $1 \leq i \leq n$, is the initial configuration of membrane i , with $l_i \in L$, the label of this membrane, $w_i \in O^*$, a finite multiset of objects and s_i , a finite set of strings over O ;
- R_i , for each $1 \leq i \leq n$, is a finite set of rewriting rules associated with membrane labelled i , having one of the following two forms:
 - *Multiset rewriting rules:*

$$obj_1 [obj_2]_l \xrightarrow{k} obj'_1 [obj'_2]_l$$

with $obj_1, obj_2, obj'_1, obj'_2 \in O^*$ some finite multisets of objects and l a label from L . A multiset of objects, obj is represented as $obj = o_1 + \dots + o_m$ with $o_1, \dots, o_m \in O$.

These multiset rewriting rules are applicable on both sides of each membrane; a multiset obj_1 which is outside a membrane l and a multiset obj_2 placed inside the same membrane can simultaneously be rewritten by a multiset obj'_1 and a multiset obj'_2 , respectively.

- *String rewriting rules:*

$$[obj_1 + str_1; \dots; obj_p + str_p]_l \xrightarrow{k} [obj'_1 + str'_{1,1} + \dots str'_{1,i_1}; \dots; obj'_p + str'_{p,1} + \dots str'_{p,i_p}]_l$$

A string str is represented as follows $str = \langle s_1.s_2.\dots.s_i \rangle$ where $s_1, \dots, s_i \in O$. In this case each multiset of objects obj_j and string str_j , $1 \leq j \leq p$, are replaced by a multiset of objects obj'_j and strings $str'_{j,1} \dots str'_{j,i_j}$.

The stochastic constant k is used to compute the propensity of the rule by multiplying it by the number of distinct possible combinations of the objects and substrings that occur on the left-side of the rule with respect to the current contents of membranes involved in the rule. The propensity associated with each rule is further utilised in generating the probability of the rule and time necessary to execute it.

Stochastic P systems are utilised to specify cellular systems consisting of molecular interactions taking place in different locations of living cells. Different regions and compartments are represented by membranes. Each molecular species is an object in the multiset associated with the region or compartment where the molecule is located. Strings are utilised to specify the genetic information encoded by DNA and RNA. Molecular interactions, compartment translocation and gene expression are specified using rewriting rules on multisets of objects and strings.

In stochastic P systems [21] constants are associated with rules in order to compute their probabilities and time needed to be applied according to Gillespie algorithm. This approach is based on a Monte Carlo algorithm for stochastic simulation of molecular interactions taking place inside a single volume or across multiple compartments [6].

In order to construct and analyse a stochastic P system model this can be translated into an adequate model checker like PRISM. This has its own language, a simple, high level, state-based language. The fundamental components of the PRISM language are modules, variables and commands. Each model is composed of a number of modules which can interact with each other. A module contains a number of local variables and commands utilised to specify certain behaviour.

The variables are utilised to keep various values that constitute the states of the module. The space of reachable states is computed using the range of each variable and its initial value. The global state of the whole model is determined by the local state of all modules.

A command defining some behaviour within a module has the following form:

$$[\text{ action }] \mathbf{g} \rightarrow \lambda_1 : \mathbf{u}_1 + \dots + \lambda_n : \mathbf{u}_n;$$

The guard \mathbf{g} is a predicate over all the variables of the model. Each update \mathbf{u}_i describes the new values of the variables in the module specifying a transition

of the module. The expressions λ_i are used to compute probabilities associated to transitions.

The label **action** placed inside the square brackets are used to synchronise different commands spread across the system. The rate of the transition resulting in this case is equal to the product of the individual rates, since the processes involved are assumed to be independent events.

There is a straightforward way of mapping stochastic P systems into PRISM representation. Compartments are mapped into modules, objects are translated as local variables, with some initial values, and rules are transcribed as transitions. For instance, a rule like

$$obj_1 [obj_2]_l \xrightarrow{k} obj'_1 [obj'_2]_l$$

where $obj_1 = obj'_1 = \lambda$ and $obj_2 = a_1 + \dots + a_p$, $obj'_2 = b_1 + \dots + b_q$, is translated into the following PRISM code given that all symbols are distinct

$$\begin{aligned} & [] \mathbf{a}_1 > \mathbf{0} \& \dots \& \mathbf{a}_p > \mathbf{0} \rightarrow \mathbf{k} : \\ & (\mathbf{a}'_1 = \mathbf{a}_1 - \mathbf{1}) \& \dots \& (\mathbf{a}'_p = \mathbf{a}_p - \mathbf{1}) \& (\mathbf{b}'_1 = \mathbf{b}_1 + \mathbf{1}) \& \dots \& (\mathbf{b}'_q = \mathbf{b}_q - \mathbf{1}). \end{aligned}$$

After an entire translation of the P system into PRISM specification language is obtained, various properties of the system can be formulated into adequate logics and simulations and verifications of certain properties can be checked.

3 P Systems Testing

All software applications, irrespective of their use and purpose, are tested before being released, installed and used. Testing is not a replacement for formal verification, it is a necessary mechanism to increase the confidence in software correctness and to make sure it works properly. Although formal verification, as previously discussed, has been applied for different P system models, testing has been neglected until [14], [16]. In the sequel it is presented a testing framework and its underpinning theory which based on formal grammars and finite state machines. We develop this testing theory based on formal grammars and finite state machines because these models of computation are the closest formalisms to P systems and testing approaches for them have been very well developed.

We will consider basic P systems in this section.

Definition 2. A P system is a tuple $\Pi = (V, \mu, w_1, \dots, w_n, R_1, \dots, R_n)$, where

- V is a finite set, called alphabet;
- μ defines the membrane structure; a hierarchical arrangement of n compartments called regions delimited by membranes; these membranes and regions are identified by integers 1 to n ;
- w_i , $1 \leq i \leq n$, represents the initial multiset occurring in region i ;
- R_i , $1 \leq i \leq n$, denotes the set of rules applied in region i .

The rules in each region have the form $a \rightarrow (a_1, t_1) \dots (a_m, t_m)$, where $a, a_i \in V$, $t_i \in \{in, out, here\}$, $1 \leq i \leq m$. When such a rule is applied to a symbol a in the current region, the symbol a is replaced by the symbols a_i which stays in this region if $t_i = here$; symbols a_i are sent to the outer region, when $t_i = out$, and symbols a_i , with $t_i = in$, are sent into one of the regions contained in the current one, arbitrarily chosen. In the following definitions and examples all the symbols $(a_i, here)$ are used as a_i , i.e., *here* destination will be removed. The rules are applied in the maximally parallel mode which means that they are used in all the regions in the same time and in each region all symbols that may be processed, must be.

A configuration of the P system Π is a tuple $c = (u_1, \dots, u_n)$, $u_i \in V^*$, $1 \leq i \leq n$. A derivation of a configuration c_1 to c_2 using the maximal parallelism mode is denoted by $c_1 \Longrightarrow c_2$. We will distinguish terminal configurations, $c = (u_1, \dots, u_n)$, as being configurations where no u_i can be further processed.

The set of all halting configurations is denoted by $L(\Pi)$, whereas the set of all configurations reachable from the initial one (including the initial configuration) is denoted by $S(\Pi)$.

Definition 3. A deterministic finite automaton (*abbreviated DFA*), M , is a tuple (A, Q, q_0, F, h) , where:

- A is the finite input alphabet;
- Q is the finite set of states;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states;
- $h : Q \times A \rightarrow Q$ is the next-state function.

The next-state function h can be extended to a function $h : Q \times A^* \rightarrow Q$ defined by:

- $h(q, \epsilon) = q$, $q \in Q$;
- $h(q, sa) = h(h(q, s), a)$, $q \in Q$, $s \in A^*$, $a \in A$.

For simplicity the same name h is used for the next-state function and for the extended function.

Given $q \in Q$, a sequence of input symbols $s \in A^*$ is said to be accepted by M in q if $h(q, s) \in F$. The set of all input sequences accepted by M in q_0 is called the *language defined (accepted) by M* , denoted $L(M)$.

3.1 Grammar-Like Testing

In *grammar engineering*, formal grammars are used to specify complex software systems, like compilers, debuggers, documentation tools, code pre-processing tools etc. One of the areas of grammar engineering is *grammar testing* which covers the development of various testing strategies for software based on grammar specifications. One of the main testing methods developed in this context refers to rule coverage, i.e., the testing procedure tries to cover all the rules of a specification [14].

In the context of grammar testing it is assumed that for a given specification defined as a grammar, an implementation of it exists and this will be tested. In order to test the implementation, a test set is built, as a finite set of sequences, that reveals potential errors. As opposed to testing based on finite state machines, where it is possible to prove that the specification and implementation, have the same behaviour, in the case of general context-free grammars this is no longer possible as it reduces to the equivalence of two such devices, which is not decidable. Of course, for specific restricted classes of context-free grammars there are decidability procedures regarding the equivalence problem and these may be considered for testing purposes as well. The best we can get is to cover as much as possible from the languages associated to the two mechanisms, and this is the role of a test set.

Although there are similarities between context-free grammars utilised in grammar testing and basic P systems, like those considered in this section, there are also major differences that pose new problems in defining testing methods and strategies. Some of the difficulties that we encounter in introducing some grammar-like testing procedures are related to: the hierarchical compartmentalisation of the entire model, parallel behaviour, communication mechanisms, the lack of a non-terminal alphabet and the use of multisets of objects instead of sets of strings.

We define some rule coverage criteria by firstly starting with one compartment P system, i.e., $\Pi = (V, \mu, w, R)$, where $\mu = [1]_1$. In the sequel, if not otherwise stated, we will consider that the specification and the implementation are given by the P systems Π and Π' , respectively. For such a P system Π , we define the following concepts.

Definition 4. A multiset denoted by $u \in V^*$, covers a rule $r : a \rightarrow v \in R$, if there is a derivation $w \Longrightarrow^* xay \Longrightarrow^* x'vy' \Longrightarrow^* u$; $w, x, y, v, u \in V^*$, $a \in V$.

Definition 5. A set $T \subseteq V^*$, is called a test set that satisfies the rule coverage (RC) criterion if for each rule $r \in R$ there is $u \in T$ which covers r .

The above criterion can be defined for terminal derivations as well and we get a new type of test set.

The following one compartment P systems are considered, $\Pi_i, 1 \leq i \leq 4$, having the same alphabet and initial multiset [14]:

$$\Pi_i = (V_i, \mu_i, w_i, R_i)$$

where

- $V_1 = V_2 = V_3 = V_4 = \{s, a, b, c\}$;
- $\mu_1 = \mu_2 = \mu_3 = \mu_4 = [1]_1$ - i.e., one compartment, denoted by 1;
- $w_1 = w_2 = w_3 = w_4 = s$;
- $R_1 = \{r_1 : s \rightarrow ab, r_2 : a \rightarrow c, r_3 : b \rightarrow bc, r_4 : b \rightarrow c\}$;
- $R_2 = \{r_1 : s \rightarrow ab, r_2 : a \rightarrow \lambda, r_3 : b \rightarrow c\}$;
- $R_3 = \{r_1 : s \rightarrow ab, r_2 : a \rightarrow bcc, r_3 : b \rightarrow \lambda\}$;
- $R_4 = \{r_1 : s \rightarrow ab, r_2 : a \rightarrow bc, r_3 : a \rightarrow c, r_4 : b \rightarrow c\}$.

In the sequel for each multiset w , we will use the following vector of non-negative integer numbers $(|w|_s, |w|_a, |w|_b, |w|_c)$.

The sets of all configurations expressed as vectors of non-negative integer numbers, computed by the P systems Π_i , $1 \leq i \leq 4$ are:

- $S(\Pi_1) = \{(1, 0, 0, 0), (0, 1, 1, 0)\} \cup \{(0, 0, k, n) | k = 0, 1; n \geq 2\}$;
- $S(\Pi_2) = \{(1, 0, 0, 0), (0, 1, 1, 0), (0, 0, 0, 1)\}$;
- $S(\Pi_3) = \{(1, 0, 0, 0), (0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 2)\}$;
- $S(\Pi_4) = \{(1, 0, 0, 0), (0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 2), (0, 0, 0, 3)\}$.

Test sets for Π_1 satisfying the RC criterion are

- $T_{1,1} = \{(0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 2)\}$ and
- $T_{1,2} = \{(0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 3)\}$,

whereas $T'_{1,1} = \{(0, 1, 1, 0), (0, 0, 0, 2)\}$ and $T'_{1,2} = \{(0, 1, 1, 0), (0, 0, 1, 2)\}$ are not, as they do not cover the rules r_3 and r_4 , respectively.

If we consider Π_1 a specification with test sets $T_{1,1}$ and $T_{1,2}$, then we observe that Π_2 fail to pass $T_{1,1}$ and $T_{1,2}$, and Π_3 fails on and $T_{1,2}$. Hence, these are faulty implementations and the errors are revealed by the test sets above. Π_4 instead, although is not correct, passes both tests. In this case a more powerful criterion is needed [14].

3.2 Finite State Machine Based Testing

We first present the process of constructing a DFA for one compartment P system. Let $\Pi = (V, \mu, w, R)$, where $\mu = [1]_1$ be such a system. In this case, the configuration of Π can change as a result of the application of some rule in R or of a number of rules, in parallel. In order to guarantee the finiteness of this process, for a given integer k , only computations of maximum k steps will be considered. For example, for $k = 4$, the tree in Figure 1 depicts all derivations in Π_1 of length less than or equal to k . The terminal nodes are in bold.

As only sequences of maximum k steps are considered, for every rule $r_i \in R$ there will be some N_i such that, in any step, r_i can be applied at most N_i times. Thus, the tree that depicts all the derivations of a P system Π with rules $R = \{r_1, \dots, r_m\}$ can be described by a DFA Dt over the alphabet $A = \{r_1^{i_1} \dots r_m^{i_m} \mid 0 \leq i_1 \leq N_1, \dots, 0 \leq i_m \leq N_m\}$, where $r_1^{i_1} \dots r_m^{i_m}$ describes the multiset with i_j occurrences of r_j , $1 \leq j \leq m$.

As Dt is a DFA over A , one can construct the minimal DFA that accepts *precisely* the language $L(Dt)$ defined by Dt . However, as only sequences of at most k transitions are considered, it is irrelevant how the constructed automaton will behave for longer sequences. Thus, a finite cover automaton can be constructed instead.

A *deterministic finite cover automaton (DFCA)* of a finite language U is a DFA that accepts all sequences in U and possibly other sequences that are longer than any sequence in U .

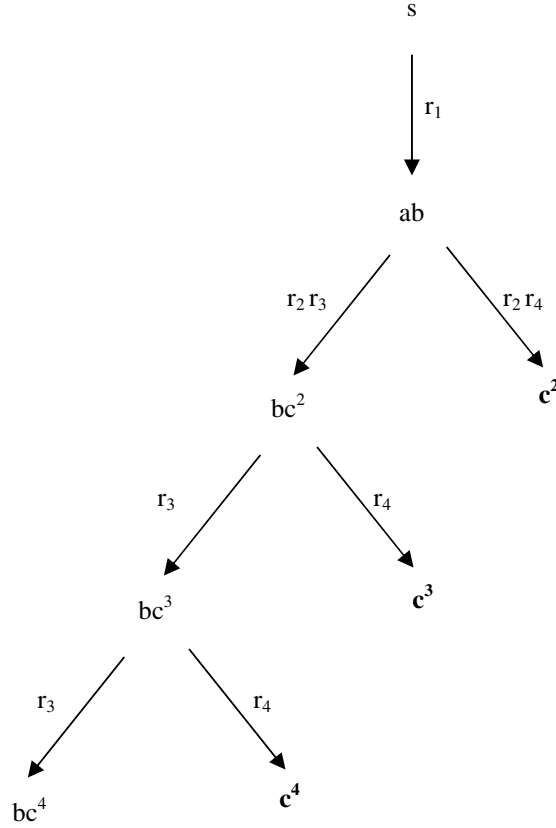


Fig. 1. Derivation tree for Π_1 and $k = 4$

Definition 6. Let $M = (A, Q, q_0, F, h)$ be a DFA, $U \subseteq A^*$ a finite language and l the length of the longest sequence(s) in U . Then M is called a deterministic finite cover automaton (DFCA) of U if $L(A) \cap A[l] = U$, where $A[l] = \bigcup_{0 \leq i \leq l} U^i$ denotes the sets of sequences of length less than or equal to l with members in the alphabet A .

A minimal DFCA of U is a DFCA of U having the least number of states. Unlike the case in which the acceptance of the precise language is required, the minimal DFCA is not necessarily unique (up to a renaming of the state space).

Any DFA that accepts U is also a DFCA of U and so the size (number of states) of a minimal DFCA of U cannot exceed the size of the minimal DFA that accepts U . On the other hand, as shown by examples in this paper, a minimal DFCA of U may have considerably fewer states than the minimal DFA that accepts U .

A minimal DFCA of the language $L(Dt)$ defined by the previous derivation tree is represented in Figure 2; q_3 in Figure 2 is final state. It is implicitly assumed that a non-final “sink” state, denoted q_s , also exists, that receives all

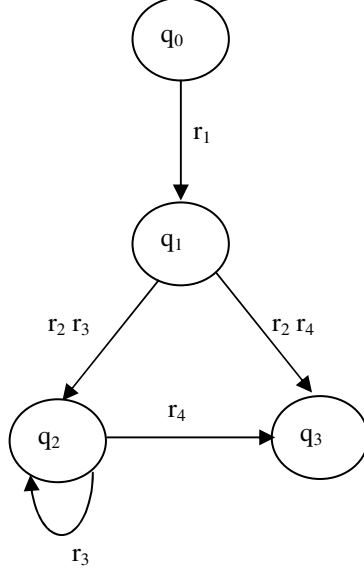


Fig. 2. Minimal DFCA for Π_1 and $k = 4$

“rejected” transitions. For testing purposes we will consider all the states as final. For details see [14].

Once the minimal DFCA $M = (A, Q, q_0, F, h)$ has been constructed, various specific coverage levels can be used to measure the effectiveness of a test set. In this paper we use two of the most widely known coverage levels for finite automata: *state coverage* and *transition coverage*.

Definition 7. A set $T \subseteq V^*$, is called a test set that satisfies the state coverage (SC) criterion if for each state q of M there exists $u \in T$ and a path $s \in A^*$ that reaches q ($h(q_0, s) = q$) such that u is derived from w through the computation defined by s .

Definition 8. A set $T \subseteq V^*$, is called a test set that satisfies the transition coverage (TC) criterion if for each state q of M and each $a \in A$ such that a labels a valid transition from q ($h(q, a) \neq q_S$), there exist $u, u' \in T$ and a path $s \in A^*$ that reaches q such that u and u' are derived from w through the computation defined by s and sa , respectively.

Clearly, if a test set satisfies TC, it also satisfies SC. A test set for Π_1 satisfying the SC criterion is

$$T_{1,1} = \{(1, 0, 0, 0), (0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 2)\},$$

whereas a test set satisfying the TC criterion is

$$T_{1,s} = \{(1, 0, 0, 0), (0, 1, 1, 0), (0, 0, 1, 2), (0, 0, 0, 2), (0, 0, 1, 3), (0, 0, 0, 3)\}.$$

The TC coverage criterion defined above is, in principle, analogous to the RC criterion given in the previous section. The TC criterion, however, does not only depend on the rules applied, but also on the state reached by the system when a given rule has been applied.

4 Conclusions

In this paper are reviewed certain aspects of formally verifying properties of some classes of P systems. Non-deterministic and stochastic classes are presented and discussed.

Testing is another investigation developed for basic classes of P systems and briefly analysed in this paper.

Acknowledgements. The research of MG and FI is supported by CNCSIS grant no.643/2009, *An integrated evolutionary approach to formal modelling and testing*.

References

1. Aguado, J., Bălănescu, T., Cowling, A., Gheorghe, M., Holcombe, M., Ipate, F.: P systems with replicated rewriting and stream X-machines (Eilenberg machines). *Fundamenta Informaticae* 49, 17–33 (2002)
2. Aman, B., Ciobanu, G.: Translating mobile ambients into P systems. *Electronic Notes in Theoretical Computer Science* 171, 11–23 (2007)
3. Andrei, O., Ciobanu, G., Lucanu, D.: Executable specifications of P systems. In: Mauri, G., Păun, Gh., Jesús Pérez-Jimenez, M., Rozenberg, G., Salomaa, A. (eds.) *WMC 2004*. LNCS, vol. 3365, pp. 126–145. Springer, Heidelberg (2005)
4. Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., Tini, S.: Compositional semantics and behavioral equivalences for P systems. *Theoretical Computer Science* 395, 77–100 (2008)
5. Andrei, O., Ciobanu, G., Lucanu, D.: A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science* 373, 163–181 (2007)
6. Bernardini, F., Gheorghe, M., Romero-Campero, R., Walkinshaw, N.: Hybrid approach to modeling biological systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) *WMC 2007*. LNCS, vol. 4860, pp. 138–159. Springer, Heidelberg (2007)
7. Cavaliere, M.: Computing by observing: A brief survey. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) *CiE 2008*. LNCS, vol. 5028, pp. 110–119. Springer, Heidelberg (2008)
8. Cavaliere, M., Mardare, R.: Partial knowledge in membrane systems: A logical approach. In: Hoogeboom, H.J., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) *WMC 2006*. LNCS, vol. 4361, pp. 279–297. Springer, Heidelberg (2006)
9. Ciobanu, G.: Semantics of P Systems. In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) *Handbook of membrane computing*, ch. 16, pp. 413–436. Oxford University Press, Oxford (to appear)
10. Dang, Z., Ibarra, O.H., Li, C., Xie, G.: Decidability of model-checking P systems. *Journal of Automata, Languages and Combinatorics* 11, 179–198 (2006)

11. Díaz-Pernil, D., Graciani, C., Gutiérrez-Naranjo, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Software for P systems. In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Handbook of membrane computing, ch. 17, pp. 437–454. Oxford University Press, Oxford (to appear)
12. Frisco, P.: P systems, Petri nets, and program machines. In: Freund, R., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2005. LNCS, vol. 3850, pp. 209–223. Springer, Heidelberg (2006)
13. Frisco, P., Corne, D.W.: Dynamics of HIV infection studied with cellular automata and conformon-P systems. *BioSystems* 91, 531–544 (2008)
14. Gheorghe, M., Ipate, F.: On testing P systems. In: Corne, D.W., Frisco, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, Springer, Heidelberg (2009)
15. Hinton, A., Kwiatkowska, M., Norman, G.: PRISM – A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
16. Ipate, F., Gheorghe, M.: Testing non-deterministic stream X-machine models and P systems. *Electronic Notes in Theoretical Computer Science* 227, 113–226 (2008)
17. Kleijn, J., Koutny, M.: Petri nets and membrane computing. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) Handbook of membrane computing, ch. 15, pp. 389–412. Oxford University Press, Oxford (to appear)
18. Păun, Gh.: Computing with membranes. *Journal of Computer and System Sciences* 61, 108–143 (2000)
19. Păun, Gh., Rozenberg, G.: A guide to membrane computing. *Theoretical Computer Science* 287, 73–100 (2002)
20. Păun, Gh.: Membrane Computing. An Introduction. Springer, Berlin (2002)
21. Pérez-Jiménez, M.J., Romero-Campero, F.: P systems, a new computational modelling tool for systems biology. In: Priami, C., Plotkin, G. (eds.) Transactions on Computational Systems Biology VI. LNCS (LNBI), vol. 4220, pp. 176–197. Springer, Heidelberg (2006)
22. Șerbănuță, T., Ștefănescu, Gh., Roșu, G.: Defining and executing P systems with structured data in K. In: Corne, D.W., Frisco, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)
23. Stamatopoulou, I., Kefalas, P., Gheorghe, M.: Transforming state-based models to P systems models in practice. In: Corne, D.W., Frisco, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 260–273. Springer, Heidelberg (2009)
24. Vitale, A., Mauri, G., Zandron, C.: Simulation of a bounded symport antiport P system with brane calculi. *Biosystems* 91, 558–571 (2008)