Model Checking Based Test Generation from P Systems Using P-Lingua

Raluca Lefticaru[1], Florentin Ipate[1], Marian Gheorghe[1,2]

[1] University of Pitesti, Department of Computer Science
Str Targu din Vale 1, 110040 Pitesti, Romania
raluca.lefticaru@gmail.com, florentin.ipate@ifsoft.ro

[2] University of Sheffield, Department of Computer Science
Regent Court, Portobello Street, Sheffield S1 4DP, UK
M.Gheorghe@dcs.shef.ac.uk

Summary. This paper presents an approach for P system testing, that uses model-checking for automatic test generation and P-Lingua as specification language. This approach is based on a transformation of the transitional, non-deterministic, cell-like P system into a Kripke structure, which is further used for test generation, by adding convenient temporal logic specifications. This paper extends our previous work in this field to multi-membrane, transitional P system, having cooperative rules, communication between membranes and membrane dissolution. A tool, which takes as input a P system specified in P-Lingua and translates it into the language accepted by the model checker NuSMV was developed and used for test case generation. Some hints regarding the automatic test generation using NuSMV and P-Lingua are also given.

1 Introduction

Membrane computing is a branch of natural computing, which investigates parallel computing models, inspired by the structure of the living cell, called *P systems*. These computational models, were introduced by Gheorghe Păun in 1998, in its seminal research report, further published as journal paper [19]. Membrane computing has known a fast growth in the last years: many variants of P systems have been proposed and results concerning their computational power and universality have been obtained. A recent handbook summarizes the most important developments in this field [21]. For all these P system variants, different implementations and simulators have been developed and consequently it appears the necessity of testing these implementations.

A first approach on testing P systems focuses on cell-like models and proposes some coverage criteria [12], which are empirically evaluated in [16]. Automatic test generation for P systems using model-checking is proposed in [15].

Given a model of a system, *model checking* [6] is a formal verification technique that explores the entire state space and decides whether this model meets a given property, expressed in temporal logic. If the property does not hold, then a counterexample is returned. This capability of model checkers to construct counterexamples provides a way to build test sets. Fraser et al. present in a recent and comprehensive survey [10] the results obtained over the last decade in software testing using model checkers.

One of the approaches used to obtain a test suite using model checking follows the steps [10]:

1. A *test purpose* is defined, describing the expected features of the test case, for example: reaching a certain state s in the model, covering a transition t, traversing a sequence of states, getting a certain value *val* of a variable x, etc.
2. These features are further specified as temporal logic properties and then converted by negation into *never-claim* conditions, or *trap properties*, such as: `G !(state = s)`, expressing that the system will never reach state s, or `G !(x = val)`, expressing that the value *val* is never taken (x is always different from *val*).
3. The model checker will verify whether the *never-claim* or *trap property* holds. If the property is false, it returns a counterexample that gives the exact path in the model that reaches state s or sets the system variable x to *val*. The counterexample will provide all the information needed to extract the test case. If the property is true, then it is impossible to build a test case satisfying the given purpose.

Regarding P system testing, one intuitive test criterion is *rule coverage*, that specifies that the test set should contain test cases which cover every rule, i.e. for each rule there exists a test case, describing a computation which involves that rule. More powerful test sets can be computed by considering the *context-dependent rule coverage* criterion. This considers coverage of rules in the context defined by other rules.

An approach on building test cases for P systems using model checking was proposed in [15]. It transforms the P system specification into a Kripke structure, then properties regarding the coverage criteria are expressed in LTL (Linear Temporal Logic) and added to the NuSMV specification. This paper extends the work from [15] in the following aspects:

- It employs the P-Lingua framework [11], to specify the P system and verify its syntactic correctness.
- It uses multi-membrane P systems, having cooperative and communication rules between membranes (the approach presented in [15] treats only one-membrane P systems, with cooperative rules).
- A transformation of P systems with *membrane dissolution* into the SMV (Symbolic Model Verifier) language is proposed.
- *Bounded model checking* is used to obtain the shortest counterexamples. This is useful in practice, to obtain a reduced test suite.
- The paper shows how other properties can be verified against the transformed model, to find possible faults in the P system.

2 Background

In the rest of the paper, we will use the following notations: V^* for the set of all strings over the alphabet $V = \{a_1, ..., a_p\}$ and λ to denote the empty string. For a string $u \in V^*$, $|u|_{a_i}$ denotes the number of a_i occurrences in u. Each string u has an associated vector of non-negative integers $(|u|_{a_1}, ..., |u|_{a_p})$. This is denoted by $\Psi_V(u)$.

2.1 P systems

A basic cell-like P system is defined as a hierarchical arrangement of membranes identifying corresponding regions of the system. Each region has associated a finite multiset of objects and a finite set of rules; both may be empty. A multiset is either denoted by a string $u \in V^*$, where the order is not considered, or by $\Psi_V(u)$. The following definition refers to one of the many variants of P systems, namely cell-like P systems, which uses transformation and communication rules [20]. We will call these processing rules. Since now onwards we will call this model P system.

Definition 1. *A* P system *is a tuple $\Pi = (V, \mu, w_1, ..., w_n, R_1, ..., R_n)$, where V is a finite set, called* alphabet*; μ defines the membrane structure, which is 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 processing rules applied in region i.*

The membrane structure, μ, is denoted by a string of left and right brackets ($[_i$, and $]_i$), each with the label of the membrane i, it points to; μ also describes the position of each membrane in the hierarchy. The rules in each region have the form $u \rightarrow (a_1, t_1)...(a_m, t_m)$, where u is a multiset of symbols from V, $a_i \in V$, $t_i \in \{in, out, here\}$, $1 \leq i \leq m$. When such a rule is applied to a multiset u in the current region, u is replaced by the symbols a_i with $t_i = here$; symbols a_i with $t_i = out$ are sent to the outer region or outside the system when the current region is the external compartment 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 when the target indication is *here*, the pair $(a_i, here)$ will be replaced by a_i. The rules are applied in maximally parallel mode.

A configuration of the P system Π, is a tuple $c = (u_1, ..., u_n)$, where $u_i \in V^*$, is the multiset associated with region i, $1 \leq i \leq n$. A computation of a configuration c_2 from c_1 using the maximal parallelism mode is denoted by $c_1 \Longrightarrow c_2$. In the set of all configurations we will distinguish terminal configurations; $c = (u_1, ..., u_n)$ is a *terminal configuration* if there is no region i such that u_i can be further developed.

We say that a rule is *cooperative* if it has at least two objects in its left hand side, e.g. $ab \rightarrow (c, in)(d, out)$. Otherwise, the rule is *non-cooperative*, e.g. $a \rightarrow (c, in)(d, out)$. The rules can also have the form $u \rightarrow v\delta$, where δ denotes the action of *membrane dissolution*: if the rule is applied, then the corresponding membrane disappears and its contents, object and membranes alike, are left free in the surrounding membrane; the rules of the dissolved membrane disappear with the membrane. The skin membrane is never dissolved. For further details regarding membrane computing, please refer to [20].

2.2 P-Lingua

P-Lingua is a programming language for membrane computing [8], developed by members of the Research Group on Natural Computing, at the University of Seville. It is developed as a free software framework for cell-like P systems and can be downloaded from http://www.p-lingua.org. Its main component is a Java library, `pLinguaCore`, that accepts as input text files (either in XML or in P-Lingua format) describing the P system model [11].

The library includes several built-in simulators for each supported model. P-Lingua 2.0 was designed for cell-like P systems and contains simulators for the following types of P systems: active membrane with division/creations rules, transition, symport/antiport, stochastic and probabilistic P systems. P-Lingua 2.1 (actual version) was extended for tissue P systems with symport/antiport rules and cell division [17].

The P-Lingua software package contains the `pLinguaCore` library and a user interface called `pLinguaPlugin`. It was used in several research papers, e.g. to solve a SAT problem using a family of P systems [8], to describe and simulate ecosystems by means of P systems [11].

A specification in P-Lingua of the P system $\Pi = (V, \mu, w_1, w_2, R_1, R_2)$, $V = \{s, a, b, c\}$, $\mu = [_1[_2]_2]_1$, $w_1 = s$, $w_2 = \lambda$, $R_1 = \{r_1 : s \rightarrow sa(b, in); r_2 : s \rightarrow ab; r3 : b \rightarrow a; r_4 : a \rightarrow c\}$, $R_2 = \{r_5 : b \rightarrow bc, r_6 : b \rightarrow c\}$ is given in Fig. 1 and can be saved in a specific file, with the `.pli` extension.

2.3 Kripke structures

Definition 2. *A Kripke structure over a set of atomic propositions AP is a four tuple $M = (S, H, I, L)$, where S is a finite set of states; $I \subseteq S$ is a set of initial states; $H \subseteq S \times S$ is a transition relation that must be left-total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in H$; $L : S \longrightarrow 2^{AP}$ is an interpretation function, that labels each state with the set of atomic propositions true in that state.*

Usually, the Kripke structure representation of a system results by giving values to every variable in each configuration of the system. Suppose $var_1, \ldots, var_n$ are the system variables, Val_i denotes the set of values for var_i and val_i is a value from Val_i, $1 \leq i \leq n$. Then the states of the system are $S = \{(val_1, \ldots, val_n) \mid val_1 \in Val_1, \ldots, val_n \in Val_n\}$, and the set of atomic predicates are $AP = \{(var_i = val_i) \mid 1 \leq i \leq n, val_i \in Val_i\}$. Naturally, L will map each state (given by the values of variables) onto the corresponding set of atomic propositions. For convenience, in

```
@model<transition>
def main()
{
      /* Initial configuration */
 @mu = [[]'2]'1;
      /* Initial multisets */
 @ms(1) = s;
 @ms(2) = #;
      /* Rules */
 [s []'2]'1 --> [s,a [b]'2]'1;
 [s --> a,b]'1;
 [b --> a]'1;
 [a --> c]'1;
 [b --> b,c]'2;
 [b --> c]'2;
}
```

Fig. 1. P-Lingua specification file for a P system with two membranes

the sequel the expressions of AP and L will not be explicitly given, the implication being that they are defined as above.

Additionally, a halt (sink) state is needed when H is not left-total and an extra atomic proposition, that indicates that the system has reached this state, is added to AP.

Definition 3. *An (infinite) path in a Kripke structure $M = (S, H, I, L)$ from a state $s \in S$ is an infinite sequence of states $\pi = s_0 s_1 \ldots$, such that $s_0 = s$ and $(s_i, s_{i+1}) \in H$ for every $i \geq 0$. A finite path π is a finite prefix of an infinite path.*

The set of all (infinite) paths from initial states is denoted by $Path(M)$. The set of all finite paths from initial states is denoted by $FPath(M)$.

2.4 Linear Temporal Logic (LTL)

The most widely used temporal specification languages in model checking are *Linear Temporal Logic* (LTL) [22, 23] and the branching time logic CTL (*Computation Tree Logic*) [5]. The superset of these logics is CTL* [9], which combines both linear-time and branching-time operators. A state formula in CTL* may be obtained from a path formula by prefixing it with a path quantifier, either an **A** (for every path) or an **E** (there exists a path).

In LTL the only path quantifier allowed is **A**, i.e. we can describe only one path properties per formula and the only state subformulas permitted are atomic propositions. More precisely, LTL formulas satisfy the following rules [6]:

- If $p \in AP$, then p is a path formula
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$ and $f\mathbf{R}g$ are path formulas, where:

- The **X** operator ("neXt time", also written $\bigcirc$) requires that a property holds in the next state of the path.
- The **F** operator ("eventually" or "in the future", also written $\Diamond$) is used to assert that a property will hold at some state on the path.
- **G** ("always" or "globally", also written $\Box$) specifies that a property holds at every state on the path.
- The **U** operator ("until") holds if there is a state on the path where g holds, and at every preceding state on the path, f holds.
- **R** ("release") is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds.

2.5 NuSMV

NuSMV is a symbolic model checker [3], developed as part of a joint project between Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST). NuSMV is the result of the reengineering, reimplementation, and, to a limited extent, extension of the SMV model checker [18], developed by CMU. It is publicly available at http://nusmv.irst.itc.it/. NuSMV [3] can process files written in SMV (Symbolic Model Verifier) language [18] (the NuSMV language is mostly source compatible with the original version of SMV) and supports LTL and CTL as temporal specification logics.

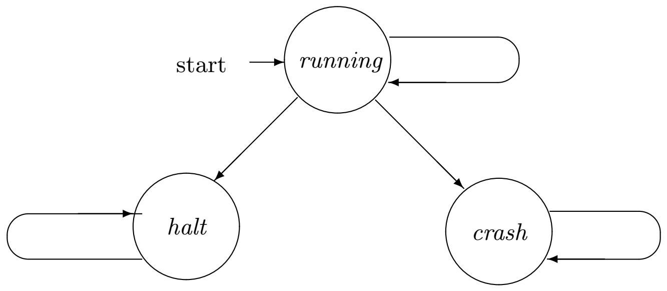

Fig. 2. Non-deterministic finite state machine

The input language of NuSMV was designed to allow descriptions of Finite State Machines (FSMs), more precisely to describe the transition relation of the FSM. This relation defines the valid evolutions of the FSM. For example, given the FSM from Fig. 2, the corresponding SMV code is:

```
MODULE main
VAR
  state : {running, halt, crash};
```

```
ASSIGN
  init(state) := running;
  next(state) := case
    state = running : {running, halt, crash};
    state = halt : halt;
    state = crash : crash;
  esac;
```

The transition relation of the FSM can be expressed also using the `TRANS` keyword. For more details refer to [3].

```
MODULE main
VAR
  state : {running, halt, crash};
ASSIGN
  init(state) := running;
TRANS
  state = running & next(state) = running |
  state = running & next(state) = halt |
  state = running & next(state) = crash |
  state = halt & next(state) = halt |
  state = crash & next(state) = crash
```

Having the model described in NuSMV, one can add LTL or CTL specifications to be verified by the model checker. For example, the specification `LTLSPEC G !( F state = halt)` is false and the counterexample obtained is the path: *running* $\rightarrow$ *halt* $\rightarrow$ *halt* $\rightarrow \ldots$ (the system will eventually remain in the *halt* state). On the other hand, the specification `LTLSPEC G !(state = halt & X state = running)` is true (there is no transition from the *halt* state, having the next state *running*).

3 Coverage criteria for P systems

A set of coverage criteria for P system rules, inspired from grammar testing, is presented in [12]. Test sets should be further designed to satisfy each coverage criterion. In the following we summarize the main coverage criteria, but for simplicity, we will provide the definitions only for one membrane P systems, $\Pi = (V, \mu, w, R)$, $\mu = [_1]_1$.

Definition 4. *A multiset denoted by* $u \in V^*$, covers *a rule* $r : a \rightarrow v \in R$, *if there is a computation* $w \Longrightarrow^* xay \Longrightarrow x'vy' \Longrightarrow^* u$; $x, y, x', y', v, u \in V^*$, $a \in V$, $w \in V^*$ *is the initial multiset. If there is no further computation from* u, *then this is called a* terminal coverage*.*

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. If every $u \in T$ provides a terminal coverage then T is called a test set that satisfies the* rule terminal coverage *(RTC) criterion.*

Definition 6. *A rule $r \in R$, $r : a \to ubv$, $u, v \in V^*$, $a, b \in V$, is called a* direct occurrence *of b. For every symbol $b \in V$, we denote by $Occs(\Pi, b)$, the set of all direct occurrences of b.*

Definition 7. *A multiset $z \in V^*$* covers *the rule $r : b \to y \in R$ for the* direct occurrence *of b, $a \to ubv \in R$, if there is a computation $w \Longrightarrow^* u_1av_1 \Longrightarrow u'_1ubvv'_1 \Longrightarrow u''_1u'yv'v''_1 \Longrightarrow^* z$; $u, v, u', v', u_1, v_1, u'_1, v'_1, u''_1, v''_1, y \in V^*$, $a, b \in V$. A set T_r is said to cover $r : b \to y$ for all direct occurrences of b if for any occurrence $o \in Occs(\Pi, b)$ there is $t \in T_r$ such that t covers r for o.*

Definition 8. *A set T is said to achieve* context-dependent rule coverage *(CDRC) for Π if it covers all $r \in R$ for all their direct occurrences. If every $z \in T$ provides a terminal coverage then T is called a test set that satisfies the* context-dependent rule terminal coverage *(CDRTC) criterion.*

To illustrate these concepts, we consider the P system $\Pi = (V, \mu, w, R)$ where: $V = \{a, b, c\}$, $\mu = [_1]_1$, $w = a$, $R = \{r_1 : a \to bc; r_2 : b \to bc; r_3 : b \to c\}$. It can be verified that the set $T = \{c^3\}$ covers all the rules, because the computation $a \Longrightarrow bc \Longrightarrow bc^2 \Longrightarrow c^3$ applies the rules r_1, r_2, r_3. Note that c^3 is a terminal configuration and, consequently, $T = \{c^3\}$ satisfies the RTC criterion. The test set $T' = \{bc^2, c^2\}$ achieves the CDRC criterion, because it covers the rules r_2, r_3, each one in the context defined by r_1. A test set satisfying the CDRTC is $T'' = \{c^2, c^3\}$.

4 Transforming a one-membrane P system into a Kripke structure

In a previous paper [15], a transformation of a one-membrane P system into a Kripke structure was proposed and several theoretical aspects were analysed. To simplify the presentation, we will consider one-membrane P system and show in next section how this approach can be extended to an arbitrary system.

Consider the one-membrane P system $\Pi = (V, \mu, w, R)$, where $R = \{r_1, \ldots, r_m\}$; each rule r_i, $1 \leq i \leq m$, is of the form $u_i \longrightarrow v_i$, where u_i and v_i are multisets over the alphabet V. In the sequel, we treat the multisets as vectors of non-negative integers, that is each multiset u is replaced by $\Psi_V(u) \in \mathbf{N}^k$, where k denotes the number of symbols in V; so, we will write $u \in \mathbf{N}^k$.

In order to define the Kripke structure associated to Π we use two predicates $MaxPar$ and $Apply$ (similar to [7]): $MaxPar(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$, $u \in \mathbf{N}^k$, $n_1, \ldots, n_m \in \mathbf{N}$ signifies that a computation from the configuration u in maximally parallel mode is obtained by applying rules $r_1 : u_1 \longrightarrow v_1, \ldots, r_m :$

$u_m \longrightarrow v_m$, $n_1, \ldots, n_m$ times, respectively (in particular, $MaxPar(u, u_1, v_1, 0, \ldots, u_m, v_m, 0)$ signifies that no rule can be applied and so u is a terminal configuration); $Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$, $u, v \in \mathbf{N}^k$, $n_1, \ldots, n_m \in \mathbf{N}$, denotes that v is obtained from u by applying rules $r_1, \ldots, r_m$, $n_1, \ldots, n_m$ times, respectively.

In order to keep the number of configurations finite, for each configuration $u = (u^{(1)}, ..., u^{(k)})$ we will assume that each component, $u^{(i)}$, $1 \leq i \leq k$ cannot exceed an established upper bound, denoted Max, and each rule can only be applied for at most a given number of times, denoted Sup.

We denote $u \leq Max$ if $u^{(i)} \leq Max$ for every $1 \leq i \leq k$ and $(n_1, \ldots, n_m) \leq Sup$ if $n_i \leq Sup$ for every $1 \leq i \leq m$; $\mathbf{N}^k_{Max} = \{u \in \mathbf{N}^k \mid u \leq Max\}$, $\mathbf{N}^m_{Sup} = \{(n_1, \ldots, n_m) \in \mathbf{N}^m \mid (n_1, \ldots, n_m) \leq Sup\}$. Analogously to [7], the system is assumed to crash whenever $u \leq Max$ or $(n_1, \ldots, n_m) \leq Sup$ does not hold (this is different from the normal termination, which occurs when $u \leq Max$, $(n_1, \ldots, n_m) \leq Sup$ and no rule can be applied). Under these conditions, the one-membrane P system Π can be described by a Kripke structure $M = (S, H, I, L)$ with $S = \mathbf{N}^k_{Max} \cup \{Halt, Crash\}$ with $Halt, Crash \notin \mathbf{N}^k_{Max}$, $Halt \neq Crash$; $I = w$ and H defined by:

- $(u, v) \in H$, $u, v \in \mathbf{N}^k_{Max}$, if $\exists (n_1, \ldots, n_m) \in \mathbf{N}^m_{Sup} \setminus \{(0, \ldots, 0)\} \cdot$
  $MaxPar(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge$
  $Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$;
- $(u, Halt) \in H$, $u \in \mathbf{N}^k_{Max}$, if $MaxPar(u, u_1, v_1, 0, \ldots, u_m, v_m, 0)$;
- $(u, Crash) \in H$, $u \in \mathbf{N}^k_{Max}$, if $\exists (n_1, \ldots, n_m) \in \mathbf{N}^m, v \in \mathbf{N}^k \cdot$
  $\neg((n_1, \ldots, n_m) \leq Sup \wedge v \leq Max) \wedge MaxPar(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge$
  $Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$;
- $(Halt, Halt) \in H$;
- $(Crash, Crash) \in H$.

It can be observed that the relation H is left-total.

The main result of [15] can be summarized by the following theorem:

Theorem 1. *Given an one-membrane P system $\Pi = (V, [_1]_1, w_1, R)$, (terminal) test suites satisfying the rule coverage and context dependent rule coverage criteria are generated based on LTL specifications.*

This means that having the transformation into a Kripke structure, what we need is to verify the following LTL specifications:

- $G\neg((n_i \geq 1) \wedge (state = other))$, for each rule $r_i \in R$, in order to achieve rule coverage (RC).
- $G\neg((n_i \geq 1) \wedge (state = other) \wedge F(state = halt))$, for each rule $r_i \in R$, in order to obtain rule terminal coverage (RTC).
- $G\neg((n_i \geq 1) \wedge X((n_j \geq 1) \wedge (state = other)))$, for each pair of rules $(r_i, r_j) \in R \times R$, where r_j can be applied the context of r_i, in order to achieve context dependent rule coverage (CDRC).

- $G\neg((n_i \geq 1) \wedge X((n_j \geq 1) \wedge (state = other)) \wedge F(state = halt))$, for each pair of rules $(r_i, r_j) \in R \times R$, where r_j can be applied the context of r_i, in order to obtain context dependent rule terminal coverage (CDRTC).

5 Generating the test suits

5.1 The test generation tool

Following the test generation strategy presented previously, a tool was developed, which functions as described by Fig. 3. A graphical interface allows editing and verification of PLI files, representing the specification of P systems in P-Lingua. The tool communicates with the P-Lingua framework, using its parser to syntactically verify the specification. Once the specification contains no syntactically errors, the corresponding objects from the library `pLinguaCore` are created. For a given `Psystem` object, an SMV file is created, containing the associated SMV model (corresponding to the Kripke structure). The user has the possibility to choose which of the following coverage criteria should be employed: RC, RTC, CDRC, CDRTC. For each coverage criteria LTL specifications, representing never-claim formulas, are added to the SMV file. Finally, the NuSMV model checker is run against the model specified in SMV, the counterexamples are decoded and transformed into test cases.

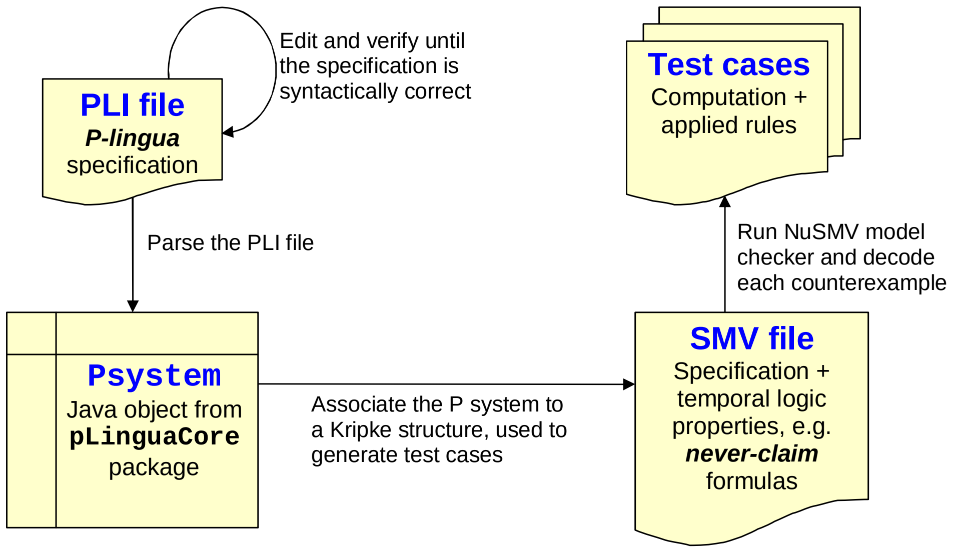
Edit and verify until the specification is syntactically correct
PLI file
P-lingua specification
Parse the PLI file
Psystem
Java object from pLinguaCore package
Associate the P system to a Kripke structure, used to generate test cases
SMV file
Specification + temporal logic properties, e.g. never-claim formulas
Run NuSMV model checker and decode each counterexample
Test cases
Computation + applied rules

Fig. 3. Tool overview

Having a P system specified in P-Lingua, the tool automatically transforms it into a SMV model. This model depends on the type of P system and in the following subsections we will present the transformation strategies proposed for different types of P systems. After the transformations the LTL specifications are automatically generated and appended to the SMV file.

5.2 Transforming one-membrane P systems into SMV

For an one-membrane P system $\Pi = (V, \mu, w, R)$, with $V = \{a_1, \ldots, a_k\}$ and $R = \{r_1, \ldots r_m\}$ (each rule r_i has the form $u_i \longrightarrow v_i$), its associated Kripke structure is $M = (S, H, I, L)$. The state space of M is implemented by using a 3-valued "state" variable (with values "Halt", "Crash" and "Running") and appropriate variables to hold the current configuration and the number of applications of each rule. Therefore, the NuSMV model will contain:

- k variables, labelled exactly like the objects from the alphabet V, each one showing the number of occurrences of each object, $a_i \in V$, $1 \leq i \leq k$;
- m variables n_i, $1 \leq i \leq m$, each one showing the number of applications of $r_i \in R$, $1 \leq i \leq m$;
- one variable *state* showing the current state of the model, *state* $\in \{Running, Halt, Crash\}$;
- two constants, Max corresponding to the upper bound for the number of occurrences expressed by each $a_i \in V, 1 \leq i \leq k$ and *Sup* which shows that each rule $r_i, 1 \leq i \leq m$, can be applied at most *Sup* times (see Section 4).

With these notations we are prepared to construct a NuSMV specification as a FSM where the states and transitions are defined below and also abstracted in Fig. 2.

If the current state is *Running* then this is characterised by the values provided by $a_1 \geq 0, \ldots, a_k \geq 0$; the maximal parallelism condition will be written as a conjunction $c_1 \wedge \cdots \wedge c_m$, where each condition c_i, $1 \leq i \leq m$, corresponds to rule r_i and is a disjunction $c_i = c_{i_1} \vee \cdots \vee c_{i_p}$, given the left hand side of r_i is $a_{i_1}^{t_{i_1}} \ldots a_{i_p}^{t_{i_p}}$. The condition c_{i_j}, $1 \leq j \leq p$, is $0 \leq a_{i_j} - n_1 h_1 - \cdots - n_m h_m < t_{i_j}$, where $n_1, \ldots, n_m$ represent the values provided by $MaxPar$ and $h_q \geq 0$ represents the number of occurrences of symbol a_{i_j} on the left hand side of r_q. This condition simply states that, after applying all rules in a maximal parallel way, the number of occurrences of symbol a_{i_j} left is less than the number of occurrences of a_{i_j} appearing on the left hand side of r_i, i.e., this rule can no longer be applied for this step. When the number of occurrences of the symbol a_{i_j} in the left side of a rule r_q is equal to 1, then the above inequality $0 \leq a_{i_j} - n_1 h_1 - \cdots - n_m h_m < t_{i_j}$ becomes $0 = a_{i_j} - n_1 h_1 - \cdots - n_m h_m$ (because $t_{i_j} = 1$).

The values $a_1 \geq 0, \ldots, a_k \geq 0$ that characterise the next state are computed as follows. Using the above notations and denoting by $next(a)$ the new value, we have $next(a_{i_j}) = a_{i_j} - n_1 h_1 - \cdots - n_m h_m + n_1 h'_1 + \cdots + n_m h'_m$, where $h'_q \geq 0$ represents the number of occurrences of symbol a_{i_j} on the right hand side of r_q.

Some additional conditions are added to the above ones in order to distinguish the destination state. These are obvious and derive from the upper bound conditions introduced. The example below illustrates the approach. We notice that all these conditions and the entire NuSMV specification, including the LTL expressions, are automatically derived from a P system using the tool developed by the authors of this paper.

We illustrate the approach by using the following one-membrane P systems: $\Pi_1 = (V_1, \mu, w_1, R_1)$, having $V_1 = \{s, a, b, c\}$, $\mu = [_1]_1$, $w_1 = s$, $R_1 = \{r_1 : s \rightarrow ab; r_2 : a \rightarrow c; r_3 : b \rightarrow bc; r_4 : b \rightarrow c\}$ and $\Pi_2 = (V_2, \mu, w_2, R_2)$, having $V_2 = \{s, a, b, c, d, x\}$, $\mu = [_1]_1$, $w_2 = s$, $R_2 = \{r_1 : s \rightarrow abc; r_2 : ab \rightarrow d^2; r_3 : c \rightarrow ab; r_4 : abd^2 \rightarrow x\}$.

The transition from the state *Running* to itself, for the P system Π_1, which has non-cooperative rules, can be written as the following NuSMV specification, where the second row shows that all the objects have been consumed and no rule can be further applied (maximal parallelism):

```
state = running & next(state) = running &
s - next(n1) = 0 & a - next(n2) = 0 & b - next(n3) - next(n4) = 0 &
next(s) = s - next(n1) &
next(a) = a - next(n2) + next(n1) &
next(b) = b - next(n3) - next(n4) + next(n1) + next(n3) &
next(c) = c + next(n2) + next(n3) + next(n4) &
! (next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) &
! (next(s) > Max | next(a) > Max | next(b) > Max | next(c) > Max |
next(n1) > Sup | next(n2) > Sup | next(n3) > Sup | next(n4) > Sup)
```

The maximal parallelism condition for Π_2, a P system with *cooperative rules*, becomes a conjunction of disjunctions $c_1 \wedge \cdots \wedge c_m$, each c_i corresponding to a rule:

```
(s-next(n1)=0) & (a-next(n2)-next(n4)=0 | b-next(n2)-next(n4)=0) &
(c-next(n3)=0) & (a-next(n2)-next(n4)=0 | b-next(n2)-next(n4)=0 |
(0<=d-2*next(n4) & d-2*next(n4)<2))
```

When one specification is false, a counterexample is given, i.e. a trace of the FSM that falsifies the property. Based on the counterexample received for the specification `G !((n1 > 0 &  X n2 > 0) & F state = halt)` of Π_1, a test sequence checking that r_2 appears in the context of r_1 on a terminal computation starting with w is obtained. This is given by $s \Longrightarrow ab \Longrightarrow c^2$ and the rules applied are r_1 first and r_2, r_4 at the second step.

In the following we will present an excerpt of a counterexample received from NuSMV, for the P system Π_1, edited for brevity:

```
-- specification G !((n3 > 0 & X n3 > 0) & F state = halt) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
```

```
-> State: 8.1 <-
  s = 1
  a = 0
  b = 0
  c = 0
  n1 = 0
  n2 = 0
  n3 = 0
  n4 = 0
  state = running
-> State: 8.2 <-
  s = 0
  a = 1
  b = 1
  n1 = 1
-> State: 8.3 <-
  a = 0
  c = 2
  n1 = 0
  n2 = 1
  n3 = 1
-> State: 8.4 <-
  c = 3
  n2 = 0
-> State: 8.5 <-
  b = 0
  c = 4
  n3 = 0
  n4 = 1
-- Loop starts here
-> State: 8.6 <-
  n4 = 0
  state = halt
```

The values of all variables are listed only once, for the first configuration of the counterexample. Then, at the following steps, only the modified variables are printed. Based on the counterexample received for the specification `G !((n3 > 0 &  X n3 > 0) & F state = halt)`, the tool computes the entire configuration at each step and the applied rules. The test case corresponding to the use of rule r_3 in the context of r_3, is represented by the P system derivation: $s \Longrightarrow ab \Longrightarrow bc^2 \Longrightarrow bc^3 \Longrightarrow c^4$. The rules used were: first r_1, then r_2, r_3, for the third transition r_3 and finally r_4, as it can be seen from the following table, corresponding to the counterexample above:

State	s	a	b	c	n1	n2	n3	n4	state
8.1	1	0	0	0	0	0	0	0	running
8.2	0	1	1	0	1	0	0	0	running
8.3	0	0	1	2	0	1	1	0	running
8.4	0	0	1	3	0	0	1	0	running
8.5	0	0	0	4	0	0	0	1	running
8.6	0	0	0	4	0	0	0	0	halt

5.3 Transforming multi-membrane P systems into SMV

The transformation of multi-membrane P systems into SMV is similar to the one for one-membrane P systems. The differences are the following:

- If the P system contains $p > 1$ membranes, the SMV model will contain $k \times p$ variables for the occurrences of the objects in each membrane; labelled like the symbols from the alphabet V, $|V| = k$, with an additional index, representing the membrane.
- The variables n_i, $1 \leq i \leq |R_1| + \ldots + |R_m|$ will be used to represent the number of applications of each rule r_i, (we have considered that the rules from all membranes are labelled $r_1, \ldots, r_{|R_1|+\ldots+|R_m|}$).

We will illustrate these differences, compared to the one-membrane P system, by specifying in NuSMV the P system $\Pi_3 = (V, \mu, w_1, w_2, R_1, R_2)$, $V = \{s, a, b, c\}$, $\mu = [_1[_2]_2]_1$, $w_1 = s$, $w_2 = \lambda$, $R_1 = \{r_1 : s \to sa(b, in); r_2 : s \to ab; r_3 : b \to a; r_4 :$

$a \to c\}$, $R_2 = \{r_5 : b \to bc, r_6 : b \to c\}$, whose P-Lingua specification is given in Fig. 1.

The SMV model has the variables: $s_1, a_1, b_1, c_1, s_2, a_2, b_2, c_2, state$, the constants Max, Sup and the differences from the previous model will be given by the communications between membranes. For example, the number of objects b in the inner membrane, labelled with b_2, will take into account: the number of b objects consumed in membrane 2 by applying rules r_5, r_6, the number of objects b produces by the rules r_5 in membrane 2 and r_1 in membrane 1. An excerpt, representing the self loop from the state *running* is the following (edited for brevity):

```
state = running & next(state) = running &
s1 - next(n1) - next(n2) = 0 & b1 - next(n3) = 0 & a1 - next(n4) = 0 &
b2 - next(n5) - next(n6) = 0 &
next(s1) = s1 - next(n1) - next(n2) + next(n1) &
next(a1) = a1 - next(n4) + next(n1) + next(n2) + next(n3) &
next(b1) = b1 - next(n3) + next(n2) &
next(c1) = c1 + next(n4) &
next(s2) = s2 &
next(a2) = a2 &
next(b2) = b2 - next(n5) - next(n6) + next(n1) + next(n5)&
next(c2) = c2 + next(n5) + next(n6) &
! (next(n1) = 0 & ... & next(n6) = 0 ) &
! (next(s1) > Max | ... | next(c1) > Max |
   next(s2) > Max | ... | next(c2) > Max |
   next(n1) > Sup | ... | next(n6) > Sup )
```

5.4 Transforming P systems with dissolving rules into SMV

Similarly to the previous section, this P system model, which has $p > 1$ membranes, will have $k \times p$ variables to represent the occurrences of the objects in each membrane, a number of variables n_i equal to the total number of rules and some special variables, to mark the membranes affected by dissolving rules.

For each membrane that can be dissolved we will consider a variable $dis_i \in \{0, 1\}$, showing whether the membrane is dissolved (1) or it is still alive (0). When the membrane is dissolved its objects are assimilated by the outer membrane. In the SMV transformation we have used the variables dis_i as flags, to correctly update the values of the variables counting the occurrences of objects in each membrane.

Consider the P system $\Pi_4 = (V, \mu, w_1, w_2, R_1, R_2)$, $V = \{a, b, c\}$, $\mu = [_1[_2]_2]_1$, $w_1 = b$, $w_2 = abc$, $R_1 = \{r_1 : b \to c; r_2 : c \to a\}$, $R_2 = \{r_3 : b \to ab; r_4 : b \to b\delta, r_5 : c \to cc\}$. The inner membrane, labelled 2, can be dissolved when the rule r_4 is applied. In the SMV model the variable dis_2 is updated by the instruction:

```
next(dis2) := case
    dis2 = 1 : 1;      -- if membrane is dissolved, it remains dissolved
    next(n4) >= 1 : 1; -- if rule r4 is applied, membrane 2 dissolves
    1 : 0;             -- otherwise, the membrane remains alive (0)
```

```
esac;
```

If the membrane is dissolved (`dis2=1`), then it remains dissolved. When rule r_4 is applied (`next(n4)>=1`) the membrane will dissolve (next value is 1). Otherwise (`1` means *true* in this case and shows the default option), the membrane will remain alive, the next value is 0 (not dissolved).

An excerpt from the SMV file, showing the transition from the *running* state to itself is given below. To ensure that, only when a membrane is dissolved its content is moved to the outer membrane, the rules for updating the quantities in the outer membrane, e.g. `next(a1)`, have added an extra term, e.g. `a2*next(dis2)`, representing the same type of objects from the inner membrane, multiplied with the next value `dis2`, because this term is null when the membrane is alive and it is exactly `a2` when the membrane dissolves. On the other hand, the values from the inner membrane will be multiplied with the opposed value, `(1-next(dis2)`. When membrane 2 dissolves, the number of objects in membrane 2 will become 0 because the factor `(1-next(dis2)` is 0. Otherwise, the factor is 1 and the value is computed as usual (subtracting the consumed objects and adding the produced ones).

```
state = running & next(state) = running &
b1 - next(n1) = 0 & c1 - next(n2) = 0 &
b2 - next(n3) - next(n4) = 0  & c2 - next(n5) = 0 &
next(a1) = a1 + next(n2) + a2*next(dis2) &
next(b1) = b1 - next(n1) + b2*next(dis2) &
next(c1) = c1 - next(n2) + next(n1) + c2*next(dis2) &
next(a2) = (a2 + next(n3))*(1-next(dis2)) &
next(b2) = (b2-next(n3)-next(n4)+next(n3)+next(n4))*(1-next(dis2)) &
next(c2) = (c2 - next(n5) + 2*next(n5))*(1-next(dis2)) &
! (next(n1) = 0 & ... & next(n5) = 0)&
! (next(a1) > Max | ... | next(c2) > Max |
   next(n1) > Sup | ... | next(n5) > Sup )|
```

5.5 Using the transformed model to verify different properties

The transformation of a P systems into a model accepted by a specific model-checker, NuSMV, was used to generate test cases. For this, LTL specifications were written, such as `G !(n_2 > 0 &  F(state = halt))`, having the purpose of obtaining a terminal coverage for rule r_2. If the LTL specification is false, the counterexample obtained is decoded: it represents a path in the SMV model, which corresponds to a (possible partial) computation in the P system. The union of all test cases will form the test suite.

If an LTL specification like `G !(n2 > 0 & F(state = halt))` is true this means that: (1) rule r_2 is never applied or (2) rule r_2 is applied, but the computation does not finish, i.e. the system does not reach the *halt* state. Verifying the simpler specification `G  !(n2 > 0)` and receiving from the model checker the response 'specification is true' reveals the fact that this rule is never applied. This is normally a fault in the model and could be obtained in situations like the following:

`@mu = [ ]'1;`	`@mu = [ ]'1;`
`@ms(1) = s;`	`@ms(1) = s;`
`[s --> a,b]'1;`	`[s --> a,b ]'1;`
`[a --> c]'1;`	`[x --> c]'1;`
`[b --> b,c]'1;`	`[b --> b,c]'1;`
`[b --> c]'1;`	`[b --> c]'1;`

In the left column is given a correct specification in P-Lingua of a certain P system; on the right side the second rule has a typo (x instead of a). Both fragments are syntactically correct, so the parser will accept them both. On the other hand the second P system has different computations and rule r_2 is never applied, fact revealed by the specification `G !(n2 > 0)`.

The automatic transformation to NuSMV allows verifications of different LTL or CTL specifications, that might be useful at designing a P system, that models a certain process. Even simple propositions like `G !(a > 100)` let us know if there exist or not a computation in which the number of objects a can reach a certain level. And this answer is obtained quickly, without simulating hundreds of computations to see if this ever happens.

5.6 Test generation using bounded model checking

Model checking tools face a combinatorial blow up of the state-space, known as the *state explosion* problem. This can occur if the system being verified has many components which can make transitions in parallel [5]. As P systems work in parallel and have a non-deterministic nature, the number of global system states may grow exponentially. One approach to alleviate this problem is based on using *Binary Decision Diagrams* (BDD) [18], this being the case of NuSMV, which implements BDD model checking.

Another approach to face the state explosion problem is to use *Bounded Model Checking* (BMC) algorithms. NuSMV provides also a BMC mode: it tries to find a counterexample of increasing length, and stops when it succeeds, declaring that the formula is false. The maximum number of iterations can be specified, the default value is 10. For testing it is preferable to obtain shorter test cases, so the BMC option can be very useful at test generation.

It should be emphasized that: if the maximum number of iterations is reached and no counterexample is found, the truth of the formula is not decided. In this case we cannot conclude that the formula is true, but only that any counter-example should be longer than the maximum length.

As a final conclusion, for test generation the BMC option of NuSMV is very useful. For verifying other properties of the P system NuSMV should be used in the default mode.

6 Related work

Only a few approaches on model checking P systems have been proposed until now. Among them, decidability of model checking problems for P systems was analysed and a discussion regarding the use of SPIN model checker provided [7], for P systems without priority rules and membrane dissolving rules. An operational semantics using rewriting logics and model checking based on Maude was given in [1]. Regarding probabilistic model checking of P systems Romero-Campero et al. proposed in [24] a transformation into a probabilistic and symbolic model checker called PRISM.

Several testing strategies for P systems have been presented, that use: coverage criteria inspired from grammar testing [12], finite state based testing [13] and stream X-machine models [14]. An approach to automate the test generation for P systems using model checking is presented in [15] and further extended in this work.

7 Conclusions

This paper extends our previous work on model-checking based P system testing [15]. It integrates this approach with P-Lingua, a software framework for cell-like P systems [11]. Compared to the previous work, the current tool offers support for multi-membrane P systems. Also, we propose an approach for transforming P systems with dissolving rules into NuSMV. The transformed model can be used not only for test generation, but also for verifying system properties. A reduction in test case size can be obtained if bounded model checking is used.

Future work will focus on other types of P systems, employed in modelling ecosystems [2], for which automatic test generation and model checking would be very useful. We will study testing and verification of probabilistic and stochastic P systems, integration with P-Lingua and possible use of a probabilistic model checker, e.g. PRISM, as suggested in [24].

Another research topic concerns the study of other model checkers and improvements made to the strategies presented, to face the *state explosion* problem, which appears when more complex models are verified.

Acknowledgements

This work was supported by CNCSIS - UEFISCSU, project number PNII IDEI 643/2008.

References

1. O. Andrei, G. Ciobanu, and D. Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373(3):163–181, 2007.
2. M. Cardona, M. A. Colomer, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A P system based model of an ecosystem of some scavenger birds. In Păun et al., editors. *Membrane Computing, 10th International Workshop, WMC 2009, Revised Selected and Invited Papers*, volume 5957 of *Lecture Notes in Computer Science*. Springer, pages 182–195, 2010.
3. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
4. G. Ciobanu, M, J. Pérez-Jiménez, and G, Păun, editors. *Applications of Membrane Computing*. Natural Computing Series. Springer, 2006.
5. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer, pages 52–71, 1981.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
7. Z. Dang, O. H. Ibarra, C. Li, and G. Xie. On the decidability of model-checking for P systems. *Journal of Automata, Languages and Combinatorics*, 11(3):279–298, 2006.
8. D. Díaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua programming environment for membrane computing. In Corne et al., editors, *Membrane Computing - 9th International Workshop, WMC 2008, Revised Selected and Invited Papers*, volume 5391 of *Lecture Notes in Computer Science*. Springer, pages 187–203, 2009.
9. E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. ACM, pages 169–180, 1982.
10. G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.
11. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0. In Păun et al. , editors. *Membrane Computing, 10th International Workshop, WMC 2009. Revised Selected and Invited Papers*, volume 5957 of *Lecture Notes in Computer Science*. Springer, pages 264–288, 2010.
12. M. Gheorghe and F. Ipate. On testing P systems. In Corne et al., editors, *Membrane Computing - 9th International Workshop, WMC 2008, Revised Selected and Invited Papers*, volume 5391 of *Lecture Notes in Computer Science*. Springer, pages 204–216, 2009.
13. F. Ipate and M. Gheorghe. Finite state based testing of P systems. *Natural Computing*, 8(4):833–846, 2009.
14. F. Ipate and M. Gheorghe. Testing non-deterministic stream X-machine models and P systems. *Electronic Notes in Theoretical Computer Science*, 227:113–126, 2009.
15. F. Ipate, M. Gheorghe, and R. Lefticaru. Test generation from P systems using model checking. *Journal of Logic and Algebraic Programming*, DOI:10.1016/j.jlap.2010.03.007, in press, 2010.

16. R. Lefticaru, M. Gheorghe, and F. Ipate. An empirical evaluation of P system testing techniques. *Natural Computing*, DOI:10.1007/s11047-010-9188-y, in press, 2010.
17. M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua based simulator for tissue P systems. *Journal of Logic and Algebraic Programming*, DOI:10.1016/j.jlap.2010.03.009, in press, 2010.
18. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.
19. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. Turku Center for Computer Science-TUCS Report 208, November 1998.
20. G. Păun. *Membrane Computing: An Introduction.* Springer-Verlag, 2002.
21. G. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
22. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.
23. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
24. F. J. Romero-Campero, M. Gheorghe, L. Bianco, D. Pescini, M. J. Pérez-Jiménez, and R. Ceterchi. Towards probabilistic model checking on P systems using PRISM. In Hoogeboom et al., editors, *Membrane Computing, 7th International Workshop, WMC 2006, Revised, Selected, and Invited Papers*, volume 4361 of *Lecture Notes in Computer Science*, pages 477–495. Springer, 2006.