Test Generation from P Systems Using Model Checking

Florentin Ipate[a,*], Marian Gheorghe[b,a], Raluca Lefticaru[a]

[a]*Department of Computer Science, University of Pitesti,*
Str. Targu din Vale 1, 110040 Pitesti, Romania
[b]*Department of Computer Science, University of Sheffield,*
Regent Court, Portobello Street, Sheffield S1 4DP, UK

Abstract

This paper presents some testing approaches based on model checking and using different testing criteria. First, test sets are built from different Kripke structure representations. Second, various rule coverage criteria for transitional, non-deterministic, cell-like P systems, are considered in order to generate adequate test sets. Rule based coverage criteria (simple rule coverage, context dependent rule coverage and variants) are defined and, for each criterion, a set of LTL (Linear Temporal Logic) formulas is provided. A codification of a P system as a Kripke structure and the sets of LTL properties are used in test generation: for each criterion, test cases are obtained from the counterexamples of the associated LTL formulas, which are automatically generated from the Kripke structure codification of the P system. The method is illustrated with an implementation using a specific model checker, NuSMV.

Keywords: P systems, Kripke structures, model checking, test generation
2000 MSC: 68Q05, 68Q60

1. Introduction

Model checking is an automated technique for verifying if a model meets a given specification, see [5]. It has been applied for checking concurrent systems, models of hardware and software designs. It starts from a model of the implementation, given as an operational specification; it also takes a temporal logic formula and verifies, through the entire state space, whether the property holds or fails. If a property violation is discovered then a counterexample is returned. Usually, these formulas describe liveness and safety requirements, such as the absence of deadlocks or other critical states that can cause the system to crash.

Testing is an essential part of software development and all software applications, irrespective of their use and purpose, are tested before being released. Testing is not a replacement for a formal verification procedure, when the former is also present, but rather a complementary mechanism to increase the confidence in software correctness [15]. In black-box (or functional) testing, the test generation is based on specification or model; if the specification or model is

[*]Corresponding author

Email addresses: `florentin.ipate@ifsoft.ro` (Florentin Ipate), `M.Gheorghe@dcs.shef.ac.uk` (Marian Gheorghe), `raluca.lefticaru@gmail.com` (Raluca Lefticaru)

expressed in a formal way, the generation process could be automated. The obtained test cases are then applied to the implementation, which is regarded as a ”black-box”.

The software testing community has used the capability of model checkers to generate test sets: the counterexamples provided by the model checkers are used to construct test cases [17], [16]. Two recent surveys on this topic are: [14] and [10]. Hierons et al. present a more general view of the interplay between testing and formal methods [14], whereas the survey of Fraser et al. describes the results obtained in the last decade in software testing using model checkers [10]. The type of testing discussed in this context is model-based testing, which assumes the existence of a model of the implementation under test, given in a certain formalism.

In order to obtain a test suite, some *test purposes* are defined, each one describing the expected characteristic of the test case (for example covering a certain state or transition in the model, traversing a sequence of states etc.). The test purpose is further specified as a temporal logic property (e.g. there exists a path in the model that reaches a certain state s) and then converted by negation to a ’never-claim’ condition (e.g. state s is never reached), see [9]. The model checker which verifies the never-claim property will produce a counterexample, if the property is false. This counterexample provides the actual test data that violates the never-claim property and also satisfies the original test purpose. When never-claim properties are based on coverage criteria, they are called *trap properties* [12]. For example, in order to create a test suite that covers all the values of a discrete variable x, a trap property for each possible value v_i, $1 \leq i \leq n$, of the variable x is needed, claiming that the value is never taken: $\mathrm{G}!(x = v_i)$ (x is always different from v_i). A different test generation approach is presented in [1]: mutated versions of the model are generated and tests cases that distinguish these mutants from the original model are automatically produced by applying model checking.

In the last ten years, a natural computing paradigm, namely *membrane computing*, has emerged as a powerful computational tool [24]. Its models, called *P systems*, have been intensively studied for their theoretical aspects as well as for various applications in biology, concurrency, graphics, and with respect to many interactions with other computational models - brane, ambient and π calculi, Petri nets, cellular automata, grammar systems [3]. Many variants have been introduced and studied, covering deterministic, non-deterministic or probabilistic phenomena. The decidability of model checking properties for P systems has also been studied in the last years [6]. A recent handbook summarizes the most important developments in the membrane computing field [26]. Apart from a strong theoretical investigation of various aspects of membrane computing, there have been many developments related to producing software applications modelled by certain P system classes. An overview of various applications and software tools developed so far, as well as of a specific programming environment called P-lingua, is provided in [26]. It is a natural question to ask whether these applications and P-lingua programs are correct or error-free. The development of certain testing strategies for these applications has relatively recently started to be studied ([13], [19], [20], [18]). These papers investigate different testing coverage principles associated with the model utilised, but do not discuss any specific method to derive test sets that will test the implementation against the model specification.

This paper applies model checking techniques to automatically generate test data for different testing criteria. First, different Kripke structures of the same model are utilised to generate *positive* and *negative* test sets and necessary conditions to generate *minimal positive test cases* are identified. Second, for cell-like P system models test sets are generated based on various rule coverage criteria. A test set satisfying the *rule coverage criterion* contains test cases that cover every rule, i.e. for each rule there exists a test case containing a computation which applies that rule. Intuitively, rule coverage is the simplest test criterion, similar to *statement coverage* in structural testing, since it ensures that each rule is applied at least once in testing. More powerful test sets can be selected by considering the *context-dependent rule coverage criterion*, in which each rule must be used in every possible context (defined by other rules). For test generation, the P system specification is first automatically transformed into a Kripke structure, which is then written in the language accepted by a specific symbolic model checker, NuSMV [2]. For each test criterion (rule coverage and context-dependent rule coverage), a set of temporal logic formulas is also automatically generated. The counterexamples produced by the model checker will provide the test cases; these are the exact P system paths, containing the configurations and the set of rules applied at each step. The paper proposes a novel approach for P systems testing, by automating the test case generation process by using model checking techniques; it also extends the test criteria previously defined to a more general context, i.e., cell-like P systems with cooperative rules.

The paper is structured as follows: Section 2 briefly presents the background notions, Section 3 shows how test cases can be generated from Kripke structures and Section 4 describes the transformation of a P system into a Kripke structure. Coverage criteria for P systems are given in Section 5; the theoretical basis for test case derivation is given in Section 6, while the practical side of the automatic generation is described in Section 7; The NuSMV implementation is illustrated with an example in Section 8. Finally, related work is reported in Section 9 and conclusions are drawn in Section 10.

2. Preliminaries

For an alphabet $V = \{a_1, ..., a_p\}$, V^* denotes the set of all strings over V; λ denotes 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. With each region there are 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 use transformation and communication rules [25]. We will call these processing rules. From now onwards we will refer to this model as simply a 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, $1 \le i \le n$*, represents the initial multiset occurring in region* i*;* R_i, $1 \le i \le n$*, denotes the set of processing rules applied in region* i*.*

The membrane structure, μ, is denoted by a string of left and right brackets ([, and]), each with the label of the membrane it points to; μ also describes the position of each membrane in the hierarchy. The rules in each region have the form $u \to (a_1, t_1)...(a_m, t_m)$, where u is a multiset of symbols from V, $a_i \in V$, $t_i \in \{in, out, here\}$, $1 \le i \le m$. 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 all the symbols $(a_i, here)$ are used as a_i. The rules are applied in maximally parallel mode which means that they are used in all the regions at the same time and in each region all the objects to which a rule *can* be applied *must* be the subject of a rule application [24].

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* from a configuration c_1 to c_2 using the maximal parallelism mode is denoted by $c_1 \Longrightarrow c_2$. Within 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.

For the type of P system we investigate in this paper, multi-membranes can be equivalently collapsed into one membrane by properly renaming symbols of the system associated with membranes. Thus, for the sake of convenience, in the remainder of this paper we will focus on P systems with only one membrane. For more details regarding different variants of P systems and their properties see [25].

2.2. 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 the sequel the expressions of AP and L will not be explicitly given, the assumption 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 $FPath(M)$.

2.3. 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*) [4]. A superset of these logics is CTL* [8], 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 **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 [5]:

- 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.
  - $\mathbf{G}f$ ("always" or "globally", also written $\Box$) specifies that a property, f, holds at every state on the path.
  - $f\mathbf{U}g$ operator (**U** means "until") holds if there is a state on the path where g holds, and at every preceding state on the path, f holds. This operator requires that f has to hold *at least* until g, which holds at the current or a future position.
  - **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. However, the first property is not required to hold eventually: if f never becomes true, g must remain true forever.

3. Test cases for Kripke structures

In this section positive and negative test cases are introduced in relation to two Kripke structures associated to a model. A necessary condition, expressed as a LTL specification, is identified for generating minimal positive test cases.

Let us assume that a system is modelled by a Kripke structure $M = (S, H, I, L)$ over AP. Let $M' = (S, H', I', L)$ be a Kripke structure over AP; M' represents the (potentially faulty) model of the implementation under test.

Definition 4. *We say that M' covers M, denoted $M \leq M'$ if $Path(M) \subseteq Path(M')$.*

Lemma 1. *M' covers M if and only if $FPath(M) \subseteq FPath(M')$.*

Proof. $\Longrightarrow$: We assume by absurd that the consequence is false and we derive the falsity of the hypothesis. If $FPath(M) \not\subseteq FPath(M')$ then there exists a finite sequence of states $\pi = s_0 \dots s_n$ such that $\pi \in FPath(M) \setminus FPath(M')$. Since the transition relation is left-total, there exists an infinite sequence $\pi' = s_0 \dots s_n s_{n+1} \dots$ such that $\pi' \in Path(M)$. Since $\pi' \notin Path(M')$, we have a contradiction.

$\Longleftarrow$: Using the same principle, we assume $Path(M) \not\subseteq Path(M')$. Then there exists an infinite sequence of states $\pi = s_0 s_1 \dots$ such that $\pi \in Path(M) \setminus Path(M')$. Let n be the minimum integer for which $\pi' = s_0 \dots s_n \notin FPath(M')$. Since $\pi' \in FPath(M)$, we have a contradiction.

Definition 5. *We say that M is trace equivalent to M', denoted $M \equiv M'$ if M' covers M and M' covers M.*

Corollary 1. *M is trace equivalent to M' if and only if $FPath(M) = FPath(M')$.*

Proof. Follows from lemma 1.

The above results suggest the following definitions of positive and negative test cases. These definitions are adapted from the concepts presented in [10]. The test sets are generated from two Kripke structures, M and M', associated with a system. The former is assumed to be correct and the other faulty.

Definition 6. *A finite sequence of states π is called a positive test case of M w.r.t. M' if $\pi \in FPath(M) \setminus FPath(M')$.*

Definition 7. *A finite sequence of states π is called a negative test case of M w.r.t. M' if $\pi \in FPath(M') \setminus FPath(M)$.*

Definition 8. *A positive test case $s_0 \dots s_n$, $n \geq 0$ is said to be* minimal *if $s_0 \dots s_{n-1}$ is not a positive test case.*

Clearly, $s_0 \dots s_n$ is a minimal test case if and only if $s_0 \dots s_n \in FPath(M) \setminus FPath(M')$ and $s_0 \dots s_{n-1} \in FPath(M) \cap FPath(M')$.

A minimal negative test case is defined similarly.

Definition 9. *Let $AP_1 = AP \cup \{(new\_var = 0), (new\_var = 1)\}$, where new_var is not a system variable and let $L_1 : (S \times \{0,1\}) \longrightarrow 2^{AP_1}$ be defined by $L_1(s,1) = L(s) \cup \{(new\_var = 1)\}$ and $L_1(s,0) = L(s) \cup \{(new\_var = 0)\}$, $s \in S$. Then $M - M'$ denotes the Kripke structure $(S \times \{0,1\}, H_1, (I \cap I') \times \{1\} \cup (I \setminus I') \times \{0\}, L_1)$, where H_1 is defined by:*

- $((s,1),(s',1)) \in H_1$ *if* $(s,s') \in H \cap H'$, $s, s' \in S$;
- $((s,1),(s',0)) \in H_1$ *if* $(s,s') \in H \setminus H'$, $s, s' \in S$;
- $((s,0),(s',0)) \in H_1$, $s, s' \in S$.

Note that, since H is left total, H_1 is also left-total.

Lemma 2. *$s_0 \dots s_n$, $n \geq 0$, is a minimal positive test case if and only if $(s_0,1)\dots(s_{n-1},1)(s_n,0) \in FPath(M - M')$.*

Proof. By induction on $k \geq 0$ it follows that $s_0 \dots s_k \in FPath(M) \cap FPath(M')$ if and only if $(s_0,1)\dots(s_k,1) \in FPath(M - M')$.

Then $s_0 \dots s_n$, $n \geq 0$, is a minimal positive test case if and only if
$s_0 \dots s_{n-1} \in FPath(M) \cap FPath(M')$ and $s_0 \dots s_n \in FPath(M) \setminus FPath(M')$ (by definition 8) if and only if
$(s_0,1)\dots(s_{n-1},1) \in FPath(M - M')$ and $(s_{n-1}, s_n) \in H \setminus H'$ if and only if
$(s_0,1)\dots(s_{n-1},1) \in FPath(M - M')$ and $(s_0,1)\dots(s_{n-1},1)\,(s_n,0) \in FPath(M - M')$ (by definition 9).

Lemma 3. *$\pi = s_0 \dots s_n$ is a minimal positive test case if and only if there exists an infinite sequence $s_{n+1}s_{n+2}\dots$ such that $(s_0,1)\dots(s_{n-1},1)(s_n,0)(s_{n+1},0)\,(s_{n+2},0)\dots \in Path(M - M')$.*

Proof. Follows from lemma 2 since H_1 is left total.

Theorem 4. *Consider the LTL specification $G(new\_var = 1)$ for a model associated with $M - M'$.*

1. *If the specification is satisfied then $FPath(M) \subseteq FPath(M')$.*
2. *If π is a counterexample, then there exists a finite prefix π' of π, $\pi' = (s_0, 1) \dots (s_{n-1}, 1)(s_n, 0)$, such that $s_0 \dots s_{n-1} s_n$ is a minimal positive test case.*

Proof. 1. Assume $FPath(M) \nsubseteq FPath(M')$. Then there exists a minimal positive test case $s_0 \dots s_n$. By lemma 3, there exists an infinite sequence $s_{n+1} s_{n+2} \dots$ such that $(s_0, 1) \dots (s_{n-1}, 1)(s_n, 0)(s_{n+1}, 0)(s_{n+2}, 0) \dots \in Path(M - M')$. This contradicts the hypothesis.

2. Suppose $(s_0, i_0)(s_1, i_1) \dots$ is a counterexample. Then there exists $n \geq 0$ such that $i_n = 0$ and $i_j = 1$ for every $j < n$. From the definition of $M - M'$ it follows that $i_j = 0$ for every $j \geq n$. By lemma 3, $s_0 \dots s_{n-1} s_n$ is a minimal positive test case.

4. Transforming a P system into a Kripke structure

This section shows how a P system can be transformed into a Kripke structure. As stated in the introduction, without loss of generality, we consider only 1-membrane P systems in order to simplify the presentation (for the type of P system used in this paper, a multi-membrane P system can be reduced to 1-membrane P system by an adequate codification of the symbols and rules).

Consider a 1-membrane P system $\Pi = (V, \mu, w, R)$, where $R = \{r_1, \dots, r_m\}$; 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 [6]): $MaxPar(u, u_1, v_1, n_1, \dots, u_m, v_m, n_m)$, $u \in \mathbf{N}^k$, $n_1, \dots, n_m \in \mathbf{N}$ signifies that a computation from the configuration u in maximally parallel mode is obtained by applying rules $r_1 : u_1 \longrightarrow v_1, \dots, r_m : u_m \longrightarrow v_m$, $n_1, \dots, n_m$ times, respectively, to u (in particular, $MaxPar(u, u_1, v_1, 0, \dots, u_m, v_m, 0)$ signifies that no rule can be applied and so u is a terminal configuration); $Apply(u, v, u_1, v_1, n_1, \dots, u_m, v_m, n_m)$, $u, v \in \mathbf{N}^k$, $n_1, \dots, n_m \in \mathbf{N}$, denotes that v is obtained from u by applying rules $r_1, \dots, r_m$, $n_1, \dots, n_m$ times, respectively.

In order to keep the number of configurations finite, we will assume that, for each configuration $u = (u^{(1)}, ..., u^{(k)})$, each component, $u^{(i)}$, $1 \leq i \leq k$, cannot exceed an established upper bound, denoted Max and, in each computation, each rule can only be applied for at most a given number of times, denoted Sup. Obviously, the existence of Max implies the existence of Sup; however, in practice it is often more convenient to explicitly impose both these constraints.

We denote $u \leq Max$ if $u^{(i)} \leq Max$ for every $1 \leq i \leq k$ and $(n_1, \dots, n_m) \leq Sup$ if $n_i \leq Sup$ for every $1 \leq i \leq m$; $\mathbf{N}^k_{Max} = \{u \in \mathbf{N}^k \mid u \leq Max\}$, $\mathbf{N}^m_{Sup} = \{(n_1, \dots, n_m) \in \mathbf{N}^m \mid (n_1, \dots, n_m) \leq Sup\}$. Analogously to [6], the system is assumed to crash whenever $u \leq Max$ or $(n_1, \dots, n_m) \leq Sup$ does not hold (this is different from the normal termination, which occurs when $u \leq Max$, $(n_1, \dots, n_m) \leq Sup$ and no rule can be applied). Under these conditions, the 1-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 $\exists (n_1, \dots, n_m) \in \mathbf{N}^m_{Sup} \setminus \{(0, \dots, 0)\} \cdot$
  $MaxPar(u, u_1, v_1, n_1, \dots, u_m, v_m, n_m) \wedge$
  $Apply(u, v, u_1, v_1, n_1, \dots, u_m, v_m, n_m)$;

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

It can be observed that the relation H is left-total. It is easy to show that for every $u, v \in \mathbf{N}^k_{Max}$, v is computed from u, in Π, if and only if $(u, v) \in H$.

5. Coverage criteria for P systems testing

In this section we define test suites which satisfy two coverage criteria: simple rule coverage and context-dependent rule coverage. The definitions given below generalize the previous definitions, given in [18], in two respects. Firstly, test cases are now considered to be sequences of vectors (multisets), not mere vectors (multisets). Secondly, cooperative rules are now considered (as opposed to [18], in which only non-cooperative rules were considered); this raises new issues, especially when defining a test set which meets the context-dependent rule coverage criterion.

Consider again a 1-membrane P system $\Pi = (V, \mu, w, R)$, and its associated Kripke structure $M = (S, H, I, L)$.

Definition 10. *A finite path $(s_0, \ldots, s_n) \in FPath(M)$, $n \ge 1$, is called a test case which covers rule r_i, $1 \le i \le m$, if there exists $p \le n-1$ for which $s_p, s_{p+1} \in \mathbf{N}^k_{Max}$ and $\exists (n_1, \ldots, n_m) \in \mathbf{N}^m_{Sup}$ such as $n_i \ge 1 \wedge MaxPar(s_p, u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge Apply(s_p, s_{p+1}, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$.*

In other words, a test case that covers a certain rule r_i is a finite sequence of P system computations in which r_i is applied at least once. A terminal test case is one which ends in a terminal configuration.

Definition 11. *A terminal test case which covers rule r_i, $1 \le i \le m$, is a test case $(s_0, \ldots, s_n)$ which covers r_i such that $MaxPar(s_n, u_1, v_1, 0, \ldots, u_m, v_m, 0)$.*

A (terminal) test suite which satisfies the simple rule coverage criterion will then be defined as a finite set of (terminal) test cases which cover all rules of the P system.

Definition 12. *A finite set $U \subseteq FPath(M)$ is called a (terminal) test suite which satisfies the simple rule coverage criterion if for every rule r_i, $1 \le i \le m$, whenever there exists a (terminal) test case which covers r_i, there exists $\pi \in U$ such that π is a (terminal) test case which covers r_i.*

A stronger criterion is context-dependent rule coverage: given rule r_j and rules $r_{i_1}, \ldots, r_{i_h}$, a test case which covers rule r_j in the context defined by rules $r_{i_1}, \ldots, r_{i_h}$ is a finite sequence of P system computations in which all rules $r_{i_1}, \ldots, r_{i_h}$ are applied one step before r_j is applied (thus providing the context for the application of r_j). A (terminal) test suite which satisfies the context dependent rule coverage criterion will then be defined as a finite set of (terminal) test cases which cover every rule in every possible context.

Note that, unlike previous work [18], the definitions below consider cooperative rules. Consequently, they use a set of rules (not a single rule like in [18]) as a context of a rule, all applied a step before the rule is used; in this case its left hand side is contained in the union of their right hand sides and is not in the right hand side of any of them.

Definition 13. *A finite path $(s_0, \ldots, s_n) \in FPath(M)$, $n \geq 2$, is called a test case which covers rule r_j in the context defined by rules $r_{i_1}, \ldots, r_{i_h}$, $1 \leq i_1, \ldots i_h, j \leq m$, $1 \leq h \leq m$, if there exists p, $p \leq n-2$ for which $s_p, s_{p+1}, s_{p+2} \in \mathbf{N}^k_{Max}$, $\exists\, (n_1, \ldots, n_m) \in \mathbf{N}^m_{Sup} \cdot n_{i_1}, \ldots, n_{i_h} \geq 1 \wedge MaxPar(s_p,$ $u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge Apply(s_p, s_{p+1}, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$ and $\exists (n'_1, \ldots, n'_m) \in \mathbf{N}^m_{Sup} \cdot n'_j \geq 1 \wedge MaxPar(s_{p+1}, u_1, v_1, n'_1, \ldots, u_m, v_m, n'_m) \wedge Apply(s_{p+1}, s_{p+2}, u_1,$ $v_1, n'_1, \ldots, u_m, v_m, n'_m)$.*

Definition 14. *A terminal test case which covers rule r_j in the context defined by rules $r_{i_1}, \ldots, r_{i_h}$, $1 \leq i_1, \ldots i_h, j \leq m$, $1 \leq h \leq m$, is a test case $(s_0, \ldots, s_n)$ which covers r_j in the context defined by $r_{i_1}, \ldots, r_{i_h}$ such that $MaxPar(s_n, u_1, v_1, 0, \ldots, u_m, v_m, 0)$.*

Definition 15. *A finite set $U \subseteq FPath(M)$ is called a (terminal) test suite which satisfies the context-dependent rule coverage criterion if for every rules $r_{i_1}, \ldots, r_{i_h}$ and r_j, $1 \leq i_1, \ldots i_h, j \leq m$, $1 \leq h \leq m$, whenever there exists a (terminal) test case which covers r_j in the context defined by $r_{i_1}, \ldots, r_{i_h}$, there exists $\pi \in U$ such that π is a (terminal) test case which covers r_j in the context defined by $r_{i_1}, \ldots, r_{i_h}$.*

6. Deriving test cases for rule coverage criteria

In this section we show how test suites which satisfy the above rule coverage criteria can be derived from the counterexamples produced by model checkers. Consider once again a 1-membrane P system $\Pi = (V, \mu, w, R)$ as above.

For $(u^{(1)}, \ldots, u^{(k)}) \in \mathbf{N}^k$ and $(n_1, \ldots, n_m) \in \mathbf{N}^m$ we denote $(u^{(1)}, \ldots, u^{(k)}) \circ (n_1, \ldots, n_m) = (u^{(1)}, \ldots, u^{(k)}, n_1, \ldots, n_m)$.

As a prerequisite, we define an additional Kripke structure $\overline{M} = (\overline{S}, \overline{H}, \overline{I}, \overline{L})$ with $\overline{S} = \mathbf{N}^k_{Max} \times \mathbf{N}^m_{Sup} \cup \{\overline{Halt}, \overline{Crash}\}$ with $\overline{Halt}, \overline{Crash} \notin \mathbf{N}^k_{Max} \times \mathbf{N}^m_{Sup}$, $\overline{Halt} \neq \overline{Crash}$; $\overline{I} = \{w \circ (0, \ldots, 0)\}$ and $\overline{H}$ defined by:

- $(u \circ (n_1, \ldots, n_m), v \circ (n'_1, \ldots, n'_m)) \in \overline{H}$, $u, v \in \mathbf{N}^k_{Max}$, $(n_1, \ldots, n_m), (n'_1, \ldots, n'_m) \in \mathbf{N}^m_{Sup}$ if $MaxPar(u, u_1, v_1, n'_1, \ldots, u_m, v_m, n'_m) \wedge$ $Apply(u, v, u_1, v_1, n'_1, \ldots, u_m, v_m, n'_m)$;
- $(u \circ (n_1, \ldots, n_m), \overline{Halt}) \in \overline{H}$, if $MaxPar(u, u_1, v_1, 0, \ldots, u_m, v_m, 0)$;
- $(u \circ (n_1, \ldots, n_m), \overline{Crash}) \in \overline{H}$, $u \in \mathbf{N}^k_{Max}$, $(n_1, \ldots, n_m), \in \mathbf{N}^m_{Sup}$ if $\exists (n'_1, \ldots, n'_m) \in \mathbf{N}^m, v \in \mathbf{N}^k$ $\cdot \neg((n'_1, \ldots, n'_m) \leq Sup \wedge v \leq Max) \wedge$ $MaxPar(u, u_1, v_1, n'_1, \ldots, u_m, v_m, n'_m) \wedge$ $Apply(u, v, u_1, v_1, n'_1, \ldots, u_m, v_m, n'_m)$;
- $(\overline{Halt}, \overline{Halt}) \in \overline{H}$;
- $(\overline{Crash}, \overline{Crash}) \in \overline{H}$.

It can be observed that the relation $\overline{H}$ is left-total.

For $x \in \mathbf{N}^k_{Max} \times \mathbf{N}^m_{Sup}$, $x = (x^1, \ldots x^k, x^{k+1}, \ldots, x^{k+m})$, we denote $proj_k(x) = (x^1, \ldots x^k)$. For a finite sequence $\pi = (x_0, \ldots, x_n)$, $x_i \in \mathbf{N}^k_{Max} \times \mathbf{N}^m_{Sup}$, $1 \leq i \leq n$, we denote $proj_k(\pi) = (proj_k(x_0), \ldots, proj_k(x_n))$.

If $\Pi = (V, \mu, w, R)$ is a 1-membrane P system and $M = (S, H, I, L)$ and $\overline{M} = (\overline{S}, \overline{H}, \overline{I}, \overline{L})$, are Kripke structures above mentioned, then $(s_0, \ldots, s_n) \in FPath(M)$ if and only if $(s'_0, \ldots, s'_n) \in$

$FPath(\overline{M})$, where $s'_i = s_i \circ (n^i_1, \ldots, n^i_m)$, and $(n^i_1, \ldots, n^i_m) \in \mathbf{N}^m$, $0 \leq i \leq n$; $((s_n = Halt)$ and $(s'_n = \overline{Halt}))$ or $((s_n = Crash)$ and $(s'_n = \overline{Crash}))$. $\overline{M}$ Kripke structure includes both M Kripke structure and the values of occurrences of each rule of the P system involved in computations. This additional information of $\overline{M}$ is utilised in the following results.

In the sequel it will be shown how various test cases are built in order to fulfill rule coverage criteria.

Suppose $(state = halt), (state = crash), (state = other) \in \overline{AP}$, $(state = halt)$ holds when $\overline{M}$ is in state $\overline{Halt}$, $(state = crash)$ holds when $\overline{M}$ is in state $\overline{Crash}$ and $(state = other)$ holds when $\overline{M}$ is neither in $\overline{Halt}$ nor in $\overline{Crash}$ state.

Lemma 5. *Consider the LTL specification* $G\neg((n_i \geq 1) \wedge (state = other))$, $1 \leq i \leq m$, *for 1-membrane P system,* Π, *with its associated Kripke structure* $\overline{M}$. *If* π *is a counterexample, then there exists a finite prefix* π' *of* π *such that* $proj_k(\pi')$ *is a test case which covers* r_i.

Proof. Let $\pi = x_0 x_1 \ldots$ be a counterexample. Then there exists p, $p \geq 1$ such that $x_p = s_p \circ (n^p_1, \ldots, n^p_m)$, with $s_p \in \mathbf{N}^k_{Max}$, $(n^p_1, \ldots, n^p_m) \in \mathbf{N}^m_{Sup}$, $n^p_i \geq 1$. From the construction of $\overline{M}$ it follows that $x_j = s_j \circ (n^j_1, \ldots, n^j_m)$, with $s_j \in \mathbf{N}^k_{Max}$, $(n^j_1, \ldots, n^j_m) \in \mathbf{N}^m_{Sup}$, for every $0 \leq j \leq p - 1$. Then $s_0 \ldots s_p$ is a test case which covers r_i.

Lemma 6. *Consider the LTL specification* $G\neg((n_i \geq 1) \wedge (state = other) \wedge F(state = halt))$, $1 \leq i \leq m$, *for 1-membrane P system,* Π, *with its associated Kripke structure* $\overline{M}$. *If* π *is a counterexample, then there exists a finite prefix* π' *of* π *such that* $proj_k(\pi')$ *is a terminal test case which covers* r_i.

Proof. Analogously to the proof of Lemma 5, there exist p, r, $1 \leq p < r$ such that $x_p = s_p \circ (n^p_1, \ldots, n^p_m)$, $x_r = s_r \circ (n^r_1, \ldots, n^r_m)$, with $s_p, s_r \in \mathbf{N}^k_{Max}$, $(n^p_1, \ldots, n^p_m), (n^r_1, \ldots, n^r_m) \in \mathbf{N}^m_{Sup}$, $n^p_i \geq 1$ and $MaxPar(s_r, u_1, v_1, 0, \ldots, u_m, v_m, 0)$; $x_j = s_j \circ (n^j_1, \ldots, n^j_m)$, with $s_j \in \mathbf{N}^k_{Max}$, $(n^j_1, \ldots, n^j_m) \in \mathbf{N}^m_{Sup}$, for every $0 \leq j \leq r - 1$. Then $s_0 \ldots s_r$ is a terminal test case which covers r_i.

Lemma 7. *Consider the LTL specification* $G\neg((n_{i_1} \geq 1) \wedge \ldots \wedge (n_{i_h} \geq 1) \wedge X((n_j \geq 1) \wedge (state = other)))$, $1 \leq i_1, \ldots, i_h, j \leq m$, $1 \leq h \leq m$, *for 1-membrane P system,* Π, *with its associated Kripke structure* $\overline{M}$. *If* π *is a counterexample, then there exists a finite prefix* π' *of* π *such that* $proj_k(\pi')$ *is a test case which covers* r_j *in the context defined by* $r_{i_1}, \ldots, r_{i_h}$.

Proof. Analogously to the proof of Lemma 5, there exists p, $p \geq 1$ such that $x_p = s_p \circ (n^p_1, \ldots, n^p_m)$, $x_{p+1} = s_{p+1} \circ (n^{p+1}_1, \ldots, n^{p+1}_m)$, with $s_p, s_{p+1} \in \mathbf{N}^k_{Max}$, $(n^p_1, \ldots, n^p_m)(n^{p+1}_1, \ldots, n^{p+1}_m) \in \mathbf{N}^m_{Sup}$, $n^p_{i_1} \geq 1, \ldots, n^p_{i_h} \geq 1$ and $n^{p+1}_j \geq 1$; $x_a = s_a \circ (n^a_1, \ldots, n^a_m)$, with $s_a \in \mathbf{N}^k_{Max}$, $(n^a_1, \ldots, n^a_m) \in \mathbf{N}^m_{Sup}$, for every $0 \leq a \leq p - 1$. Then $s_0 \ldots s_p s_{p+1}$ is a test case which covers r_j in the context defined by $r_{i_1}, \ldots, r_{i_h}$.

Lemma 8. *Consider the LTL specification* $G\neg((n_{i_1} \geq 1) \wedge \ldots \wedge (n_{i_h} \geq 1) \wedge X((n_j \geq 1) \wedge (state = other) \wedge F(state = halt)))$, $1 \leq i_1, \ldots, i_h, j \leq m$, $1 \leq h \leq m$, *for 1-membrane P system,* Π, *with its associated Kripke structure* $\overline{M}$. *If* π *is a counterexample, then there exists a finite prefix* π' *of* π *such that* $proj_k(\pi')$ *is a terminal test case which covers* r_j *in the context defined by* $r_{i_1}, \ldots, r_{i_h}$.

Proof. Follows the lines of the above proofs.

Theorem 9. *Given a 1-membrane P system, (terminal) test suites satisfying the rule coverage and context-dependent rule coverage criteria are generated based on LTL specifications for every rule of P.*

The generation of these test suites follows from the above lemmas.

As previously mentioned, multi-compartment P systems of the type presented in this paper can be converted to 1-compartment P systems. On the other hand, the approach introduced for P systems with one single compartment can be generalised (and therefore directly applied) to multiple compartments using the definitions and techniques given in [18]. This generalisation may be useful in practice, if we need to keep the system as it is originally specified.

7. Generating the test suits

For automatic generation of test suits we have used NuSMV, a symbolic model checker [2], publicly available at http://nusmv.irst.itc.it/. NuSMV can process files written in SMV language [21] and supports LTL and CTL as temporal specification logics.

For a 1-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 $M = (S, H, I, L)$. This is represented in NuSMV as follows. 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, the upper bound for the number of occurrences of each $a_i \in V, 1 \leq i \leq k$, and Sup, the upper bound for the number of applications of each rule r_i, $1 \leq i \leq m$ (see Section 4).

We can now construct a NuSMV specification as a Finite State Machine (FSM) whose states and transitions are defined below.

If the current state is *Running* then the current configuration 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 $c_1 \wedge \ldots \wedge c_m$, where each condition c_i, $1 \leq i \leq m$, corresponds to rule r_i and is a disjunction $c_i = c_{i_1} \vee \ldots \vee c_{i_p}$, given that 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 $0 \leq a_{i_j} - n_1h_1 - \ldots - n_mh_m < t_{i_j}$, 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 and so this rule can no longer be applied in this step. When the number of occurrences of the symbol a_{i_j} on the left side of a rule r_q is equal to 1, the above inequality $0 \leq a_{i_j} - n_1h_1 - \ldots - n_mh_m < t_{i_j}$ becomes $0 = a_{i_j} - n_1h_1 - \ldots - n_mh_m$ (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 notation and denoting by $next(a)$ the new value, we have $next(a_{i_j}) = a_{i_j} - n_1h_1 - \ldots - n_mh_m + n_1h'_1 + \ldots + n_mh'_m$, 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 those given above, in order to distinguish the destination state. These are obvious and derive from the upper bound conditions introduced. The example below illustrates the approach. Note that all these conditions and the entire NuSMV specification, including the LTL expressions, are automatically derived from a P system by using a tool developed by the authors of this paper.

8. NuSMV implementation - example

We illustrate the approach by using the following 1-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, in which 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 $c_1 \wedge \ldots \wedge c_m$, 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 $\{s\}\{ab\}\{c^2\}$ and the rules applied are first r_1 and then r_2, r_4, at the next step. Similarly, the LTL specification `G !(((n2 > 0 & n3 > 0) & X n4 > 0) & F state = halt)` of Π_2 has a corresponding test case: $\{s\}\{abc\}\{abd^2\}\{x\}$ and the rules applied in the computation $s \Longrightarrow abc \Longrightarrow abd^2 \Longrightarrow x$ are first r_1, then r_2, r_3 and, finally, r_4, at the last step. The complete NuSMV specification of Π_1 is given in Appendix A and a counterexample returned by NuSMV with its corresponding test case is shown in Appendix B.

9. Related work

P system testing has been studied now for a few years and three main directions have been considered: finite state machine based testing [13, 18], mutation based testing [19], and testing based on model checking. The first and third directions are somehow related since the underlying model used in both cases is a state-transition system. Furthermore, coverage based criteria can be used for test generation in both cases. In finite state machine based testing, however, conformance methods [18] (which ensure that the (unknown) implementation under test conforms to the specification, provided its model belongs to a certain, pre-defined, set of finite state machines, which makes up the so-called *fault model* of the method in question). Both approaches (finite state machine based and model checking based) may suffer from the state explosion associated with the construction of the state-transition model of the P system. On the other hand, in NuSMV (as well as other existing model checkers), heuristics are available for partially controlling the state explosion [2]. In any case, further case studies based on P system models of realistic systems are needed to properly assess the strengths and weaknesses of each approach.

No implementation of mutation based testing of P systems exists yet. In future research it may be interesting to consider mixed approaches, in which mutation analysis is applied to state-transition models of P Systems.

10. Conclusions

This paper, first, discusses a general testing strategy derived from Kripke structure representations of the same system using model checking and, second, introduces a testing methodology for P systems based on rule coverage criteria and model checking. The methodology presented is implemented using the symbolic model checker NuSMV and is applied to a basic class of 1-membrane P systems with cooperative rules, but it is also applicable to multi-membrane P systems. Furthermore, it can be generalized to other classes of P systems and can use other model checkers, a topic we will follow in future papers.

The coverage criteria used in this paper have been introduced and studied for P systems in [18], but here we consider systems with non-cooperative rules and generate, as test suites, sequences of multisets rather than just multisets. Additionally, the entire NuSMV specification is automatically obtained from the P system. This aspect is important when using various P system based specification languages, like P-lingua, [7, 11], since it enables the direct derivation of test sets, satisfying given coverage criteria, from the specification.

Further work will concentrate on more complex types of P systems, integrating this approach with P-lingua and on LTL rewriting to improve the performance of test generation method.

Acknowledgments

This work was supported by CNCSIS - UEFISCSU, project number PNII - IDEI 643/2008. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

Appendix A. NuSMV Specification

Based on the P system specification, the tool developed by the authors generates an SMV file, that will be processed by the NuSMV model checker. The result received from NuSMV will be further parsed by the tool, each counterexample will be 'decoded' and the corresponding test case will be obtained. In the sequel is given an SMV file, automatically generated for the P system Π_1, having one membrane $\mu = [_1]_1$, the alphabet $V_1 = \{s, a, b, c\}$, the initial multiset $w_1 = s$ and the set of rules $R_1 = \{r_1 : s \to ab; r_2 : a \to c; r_3 : b \to bc; r_4 : b \to c\}$. To increase the readability of the SMV code, we have added some comments, introduced by "--".

```
MODULE main

VAR
s : 0..10;
a : 0..10;
b : 0..10;
c : 0..10;
n1 : 0..10;
n2 : 0..10;
n3 : 0..10;
n4 : 0..10;
state : {running, halt, crash};

DEFINE
Max := 5;
Sup := 5;

ASSIGN
init(s) := 1;
init(a) := 0;
init(b) := 0;
init(c) := 0;
init(n1) := 0;
init(n2) := 0;
init(n3) := 0;
init(n4) := 0;
init(state) := running;

-- Initially the system is in the 'running' state and has only one
-- object 's'. The possible transitions are given in the TRANS
-- section and these are: running -> {running, crash, halt},
-- halt -> halt and crash -> crash.

TRANS

-- The system remains in the 'running' state. Then, conditions for
-- maximal parallelism are expressed: no further rule can be applied,
-- e.g. all the "b" objects, used by the rules r3 and r4, are
-- consumed: b-next(n3)-next(n4)=0. The 'next' values for s, a, b, c
-- are set, taking into account their previous values, the objects
-- produced and consumed by the rules. Non halting and non crashing
-- conditions follow.

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 system enters in the 'halt' state. The main difference from the
-- previous transition 'running -> running' is given by the halting
-- condition: next(n1)=0 & next(n2)=0 & next(n3)=0 & next(n4)=0

state = running & next(state) = halt &
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  ) |

-- The system enters in the 'crash' state. The main difference from the
-- transition 'running -> running' is given by the crashing rule:
-- next(s)>Max | ... | next(c)>Max | next(n1)>Sup | ... | next(n4)>Sup

state = running & next(state) = crash &
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) |

-- Loop 'halt -> halt': the variables will keep their previous values
state = halt & next(state) = halt &
(next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) &
next(s) = s & next(a) = a & next(b) = b & next(c) = c |

-- Loop 'crash -> crash'
state = crash & next(state) = crash &
next(n1) = n1 & next(n2) = n2 & next(n3) = n3 & next(n4) = n4 &
next(s) = s & next(a) = a & next(b) = b & next(c) = c

-- Simple integrity checks
LTLSPEC G ( state = running -> (0 <= s & s <= Max) );
LTLSPEC G ( state = running -> (0 <= a & a <= Max) );
LTLSPEC G ( state = running -> (0 <= b & b <= Max) );
LTLSPEC G ( state = running -> (0 <= c & c <= Max) );
LTLSPEC G ( state = running -> (0 <= n1 & n1 <= Sup) );
LTLSPEC G ( state = running -> (0 <= n2 & n2 <= Sup) );
LTLSPEC G ( state = running -> (0 <= n3 & n3 <= Sup) );
LTLSPEC G ( state = running -> (0 <= n4 & n4 <= Sup) );
LTLSPEC G ( state = halt -> ( n1 = 0 & n2 = 0 & n3 = 0 & n4 = 0 ) );
LTLSPEC G ( state = crash -> ( n1 > Sup | n2 > Sup | n3 > Sup |
n4 > Sup | s > Max | a > Max | b > Max | c > Max ) );

-- LTL specifications for Rule Coverage (RC) express that the rule
-- is never applied, in order to obtain a counterexample
LTLSPEC G !( n1 > 0 );
LTLSPEC G !( n2 > 0 );
LTLSPEC G !( n3 > 0 );
LTLSPEC G !( n4 > 0 );

-- LTL specifications for Rule Terminal Coverage (RTC) are similar to
-- RC specifications, having in addition the condition F(state = halt)
LTLSPEC G !( n1 > 0 & F(state = halt) );
LTLSPEC G !( n2 > 0 & F(state = halt) );
LTLSPEC G !( n3 > 0 & F(state = halt) );
LTLSPEC G !( n4 > 0 & F(state = halt) );

-- LTL specifications for Context Dependent Rule Coverage (CDRC)
LTLSPEC G !( n1 > 0 & X(n2 > 0) );
LTLSPEC G !( n1 > 0 & X(n3 > 0) );
LTLSPEC G !( n1 > 0 & X(n4 > 0) );
LTLSPEC G !( n3 > 0 & X(n3 > 0) );
LTLSPEC G !( n3 > 0 & X(n4 > 0) );

-- LTL specifications for Context Dependent Terminal Rule Coverage,
-- or CDRTC, are similar to CDRC, having in addition F(state = halt)
LTLSPEC G !( n1 > 0 & X(n2 > 0) & F(state = halt) );
LTLSPEC G !( n1 > 0 & X(n3 > 0) & F(state = halt) );
LTLSPEC G !( n1 > 0 & X(n4 > 0) & F(state = halt) );
LTLSPEC G !( n3 > 0 & X(n3 > 0) & F(state = halt) );
LTLSPEC G !( n3 > 0 & X(n4 > 0) & F(state = halt) );
```

Appendix B. Counterexample and corresponding test case

This is an excerpt of the counterexample received from NuSMV, using the above model, for the LTL specification, `G !((n3 > 0 & X n3 > 0) & F state = halt)`, edited for brevity:

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

```
-> State: 17.1 <-
  s = 1
  a = 0
  b = 0
  c = 0
  n1 = 0
  n2 = 0
  n3 = 0
  n4 = 0
  state = running
-> State: 17.2 <-
  s = 0
  a = 1
  b = 1
  n1 = 1
-> State: 17.3 <-
  a = 0
  c = 2
  n1 = 0
  n2 = 1
  n3 = 1
-> State: 17.4 <-
  c = 3
  n2 = 0
-> State: 17.5 <-
  b = 0
  c = 4
  n3 = 0
  n4 = 1
-- Loop starts here
-> State: 17.6 <-
  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 developed 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 computation: $s \Longrightarrow ab \Longrightarrow bc^2 \Longrightarrow bc^3 \Longrightarrow c^4$. The rules used are: 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
17.1	1	0	0	0	0	0	0	0	running
17.2	0	1	1	0	1	0	0	0	running
17.3	0	0	1	2	0	1	1	0	running
17.4	0	0	1	3	0	0	1	0	running
17.5	0	0	0	4	0	0	0	1	running
17.6	0	0	0	4	0	0	0	0	halt

References

[1] P.E. Ammann, P.E. Black, W. Majurski, Using model checking to generate tests from specifications, in: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, ICFEM '98, IEEE Computer Society, 1998, p. 46.

[2] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model checker, International Journal on Software Tools for Technology Transfer 2 (2000) 410–425.

[3] G. Ciobanu, M.J. Pérez-Jiménez, G. Păun (Eds.), Applications of Membrane Computing, Natural Computing Series, Springer, 2006.

[4] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, 1981, in: D. Kozen (Ed.), Logic of Programs, Workshop, volume 131 of *Lecture Notes in Computer Science*, Springer, 1982, pp. 52–71.

[5] E.M. Clarke, Jr., O. Grumberg, D.A. Peled, Model checking, MIT Press, Cambridge, MA, USA, 1999.

[6] Z. Dang, O.H. Ibarra, C. Li, G. Xie, On the decidability of model-checking for P systems, Journal of Automata, Languages and Combinatorics 11 (2006) 279–298.

[7] D. Díaz-Pernil, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, A P-lingua programming environment for membrane computing, in: D.W. Corne, P. Frisco, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing - 9th International Workshop, WMC 2008, Revised Selected and Invited Papers, volume 5391 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 187–203.

[8] E.A. Emerson, J.Y. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, in: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC '82, ACM, 1982, pp. 169–180.

[9] A. Engels, L.M.G. Feijs, S. Mauw, Test generation for intelligent networks using model checking, in: E. Brinksma (Ed.), Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, volume 1217 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 384–398.

[10] G. Fraser, F. Wotawa, P. Ammann, Testing with model checkers: a survey, Software Testing, Verification and Reliability 19 (2009) 215–261.

[11] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, An overview of P-Lingua 2.0, in: G. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing - 10th International Workshop, WMC 2009, Revised Selected and Invited Papers, volume 5957 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 264–288.

[12] A. Gargantini, C.L. Heitmeyer, Using model checking to generate tests from requirements specifications, in: O. Nierstrasz, M. Lemoine (Eds.), Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, volume 1687 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 146–162.

[13] M. Gheorghe, F. Ipate, On testing P systems, in: D.W. Corne, P. Frisco, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing - 9th International Workshop, WMC 2008, Revised Selected and Invited Papers, volume 5391 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 204–216.

[14] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapour, P. Krause, G. Luettgen, A. Simons, S. Vilkomir, M. Woodward, H. Zedan, Using formal specifications to support testing, ACM Compt. Surv. 41 (2009).

[15] M. Holcombe, F. Ipate, Correct Systems: Building a Business Process Solution, Applied Computing Series, Springer-Verlag, Berlin, Germany, 1998.

[16] H.S. Hong, I. Lee, O. Sokolsky, S.D. Cha, Automatic test generation from statecharts using model checking, in: In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of *BRICS Notes Series*, pp. 15–30.

[17] H.S. Hong, I. Lee, O. Sokolsky, H. Ural, A temporal logic based theory of test coverage and generation, in: J.P. Katoen, P. Stevens (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, volume 2280 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 327–341.

[18] F. Ipate, M. Gheorghe, Finite state based testing of P systems, Natural Computing 8 (2009) 833–846.

[19] F. Ipate, M. Gheorghe, Mutation based testing of P systems, International Journal of Computers, Communications & Control 4 (2009) 253–262.

[20] F. Ipate, M. Gheorghe, Testing non-deterministic stream x-machine models and p systems, Electronic Notes in Theoretical Computer Science 227 (2009) 113–126.

[21] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publ., 1993.

[22] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, IEEE, 1977, pp. 46–57.

[23] A. Pnueli, The temporal semantics of concurrent programs, Theoretical Computer Science 13 (1981) 45–60.

[24] G. Păun, Computing with membranes, Journal of Computer and System Sciences 61 (2000) 108–143.

[25] G. Păun, Membrane Computing: An Introduction, Springer-Verlag, 2002.

[26] G. Păun, G. Rozenberg, A. Salomaa (Eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, 2010.