An Integrated Approach to P Systems Formal Verification

Marian Gheorghe[1,2], Florentin Ipate[2],
Raluca Lefticaru[2], and Ciprian Dragomir[1]

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

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

Abstract. This paper presents a method to formally verify P system specifications by first identifying invariants and then checking them, using the NuSMV model checker, against a Kripke structure representation. The method is applied to a basic class of P systems with transformation and communication rules using either maximal parallelism or asynchronous rewriting strategy and for a special variant of P systems with electrical charges, but without active membranes.

1 Introduction

P systems, introduced in [19], represent a new computational model inspired by the structure and functioning of the living cell. In the last ten years there have been various investigations related to this computational paradigm, ranging from computability and complexity for different variants of these systems [20] to various applications and connections with other computational models [4]. In the last period the research on various programming approaches to P systems ([8], [22]) and formal semantics ([3], [1], [15]), or with respect to decidability of some model checking properties [7], has created the basis for investigating different aspects related to the formal verification and testing of these systems.

The testing has been investigated for some covering criteria ([11], [13]) and certain formal based approaches [14].

Formal verification has been studied for different variants of P systems by using rewriting logic and the Maude tool [1] or, for stochastic systems [2], PRISM and associated probabilistic temporal logic [12].

In this paper we consider an integrated method for formally verifying P systems. A method for identifying invariants in a formal specification and a tool, NuSMV, that checks such properties against a Kripke structure representation is presented. This method is applied for a basic class of P systems, with transformation and communication rules using either maximal parallelism or asynchronous rewriting strategy and for a special variant of P systems with electrical charges, but without active membranes. The invariants are extracted from traces of simulations of P systems represented in P-Lingua.

M. Gheorghe et al. (Eds.): CMC 2010, LNCS 6501, pp. 226–239, 2010.

2 Basic Definitions and Preliminary Relationships

2.1 P Systems

A basic cell-like P system is defined as a hierarchical arrangement of compartments delimited by membranes. Each compartment may contain a finite multiset of objects and a finite set of rules, as well as a finite set of other compartments. The rules perform transformation and communication operations. The class of such models will be called *transformation-communication P systems.*

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, i.e., the hierarchical arrangement of n compartments called* regions *delimited by* membranes*; the 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 ([, and]), each with the label of the membrane it points to and describing the position of this 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, $a_i \in V$, $t_i \in \{in, out, here\}$, $1 \leq i \leq m$. When such a rule is applied to a multiset u in the current compartment, u is replaced by the symbols a_i. The symbols a_i with $t_i = here$, remain in the compartment; if $t_i = out$, then they are sent to the outer compartment or outside the system when the current compartment is the external one; when $t_i = in$, the symbols are sent into one of the compartments 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 compartments at the same time and in each compartment all the objects to which a rule *can* be applied it *must* be the subject of a rule application [19].

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 compartment i, $1 \leq i \leq n$. A *computation* from a configuration c_1 to c_2 using the maximal parallelism mode is denoted by $c_1 \Longrightarrow c_2$.

A configuration, $c = (u_1, ..., u_n)$, is a *terminal configuration* if there is no compartment i such that u_i can be further developed.

Another variant of P systems considered in this paper will be the *P systems with electrical charges.* This is a simplification of the usual variant occurring in the literature [21]. Each compartment has a specific electrical charge $(+, -, 0)$ which can be changed by a communication rule. The set of electrical charges is denoted by H. The set of rules contains the following types of rules:

- $[u \to v]_b^h$;
- $u[]_b^{h_1} \to [v]_b^{h_2}$;
- $[u]_b^{h_1} \to v[]_b^{h_2}$;

where b indicates a compartment and $h, h_1, h_2 \in H$. The rules are applied in the normal way; for more details see [21].

The maximal parallelism can be replaced by other execution strategies. One of them, called *asynchronous execution mode*, implies that at each step at least one rule is executed.

In the sequel we will consider transformation-communication P systems using maximal parallelism or with asynchronous behaviour or P systems with electrical charges and maximal parallelism.

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.*

In general, the Kripke structure representation of a system consists of sets of values associated to the system variables. Assuming that $var_1, \dots, var_n$ are the system variables and Val_i the set of values for var_i, with val_i a value from Val_i, $1 \le i \le n$, we can introduce the states of the system as

$$S = \{(val_1, \dots, val_n) \mid val_1 \in Val_1, \dots, val_n \in Val_n\}.$$

The set of atomic predicates are given by $AP = \{(var_i = val_i) \mid 1 \le i \le n, val_i \in Val_i\}$. Naturally, L will map each state (given by the values of system variables) onto the corresponding set of atomic propositions.

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.

2.3 Linear Temporal Logic (LTL)

The most widely used query languages in model checking are based on *Linear Temporal Logic* (LTL) [17,18] and the branching time logic CTL (*Computation Tree Logic*) [5]. A 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 **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 property 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") requires that a property holds in the next state of the path.
- The **F** operator ("eventually" or "in the future") is used to assert that a property will hold at some state on the path.
- $\mathbf{G}f$ ("always" or "globally") 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.

2.4 Transformation-Communication P Systems and Kripke Structure

In this section, following the presentation from [14], it is shown how a P system operating in a maximal parallel manner can be transformed into a Kripke structure. Then this will be adapted for other types of P systems. We only consider 1-membrane P systems in order to simplify the presentation. The approach presented below can be generalised for membrane systems with arbitrary number of compartments.

Consider a 1-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. If k denotes the number of symbols in V and u a multiset of V, then we will write $u \in \mathbf{N}^k$.

The Kripke structure associated to Π utilises two predicates, $MaxPar$ and *Apply* (similar to [7]):

$$MaxPar(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m), u \in \mathbf{N}^k, n_1, \ldots, n_m \in \mathbf{N},$$

$$Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m), u, v \in \mathbf{N}^k, n_1, \ldots, n_m \in \mathbf{N}.$$

The first predicate shows that a computation from the configuration u in maximally parallel mode is obtained by applying the rules $r_1 : u_1 \longrightarrow v_1, \ldots, r_m : u_m \longrightarrow v_m$, $n_1, \ldots, n_m$ times, respectively, to u; 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.

The predicate *Apply* 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, 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.

We denote $u \leq Max$ whenever $u^{(i)} \leq Max$ for every $1 \leq i \leq k$ and similarly $(n_1, \ldots, n_m) \leq Sup$ 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 1-membrane P system Π can be described by a Kripke structure $M_\Pi = (S, H, I, L)$ 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. 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 $(u, v) \in H$, hence Π and M_Π show the same behaviour.

In the rest of our presentation we will consider a more compact form of the Kripke structure, M_Π, whereby all the states $\mathbf{N}^k_{Max}$ will be replaced by one *Running* state. So, *Running* state will define a state of normal behaviour as opposed to the situation described by the other two states, *Halt* and *Crash*. In the first case the system stops in normal circumstances whereas in the case of the *Crash* state, the system fails by going beyond some initially set finite limits.

Note that, in this section, the Kripke structure representation of a P system is given for maximal parallelism. On the other hand, the associated Kripke structure of a P system can be similarly constructed for other execution modes as well (e.g. asynchronous rewriting strategy) as illustrated by the examples given in the next section.

3 Transforming P Systems to NuSMV Specifications

In this section it is shown how the P systems considered in this paper will be mapped into the NuSMV model checker by adequately codifying Kripke structures associated with them in accordance with the principles presented in [14] and the description made in Section 2.4. Simple examples will illustrate the presentation.

3.1 Transformation-Communication P Systems to NuSMV Specifications

The presentation will follow the general principles, introduced in [14], for translating such P systems into NuSMV. The presentation below will be illustrated by the following example: $\Pi_1 = (V, []_1, w_1, R_1)$, where $V = \{a, b, c, d, x, y\}$, $w_1 = xy$, $R_1 = \{r_1 : x \rightarrow a, r_2 : y \rightarrow b, r_3 : a \rightarrow xc, r_4 : b \rightarrow ydd\}$. Please note that the system will not halt, but this is less significant in this context. A computation in Π_1 has the following steps

$$xy \Longrightarrow ab \Longrightarrow xcydd \Longrightarrow acbdd \Longrightarrow xccydddd \ldots \Longrightarrow xc^nyd^{2n} \Longrightarrow ac^nbd^{2n} \ldots$$

In the NuSMV codification, we will use for each symbol, a, $a \in V$, above, a variable with the same name to denote the number of occurrences of this symbol in compartment 1 in the current step. If more than a compartment is considered then this variable should be indexed by the compartment number. For each rule r_i, we will identify by `ni` the NuSMV variable that expresses the number of times r_i is applied in the current step in a maximally parallel manner - this is the value that appears in the $MaxPar$ predicate.

We will describe a computation step in NuSMV by a transition from the state *Running* to itself in the Kripke structure associated with Π_1, M_{Π_1}. The maximal parallelism is expressed by the condition

```
x - next(n1) = 0 & y - next(n2) = 0 & a - next(n3) = 0 & b - next(n4) = 0
```

Additional conditions to characterise the *Running* state as well as equations to compute the values of the multisets in the next step are provided below.

```
state = running & next(state) = running &
x - next(n1) = 0 & y - next(n2) = 0 & a - next(n3) = 0 & b-next(n4) = 0 &
next(x) = x - next(n1) + next(n3) &
next(y) = y - next(n2) + next(n4) &
next(a) = a - next(n3) + next(n1) &
next(b) = b - next(n4) + next(n2) &
next(c) = c + next (n3) &
next(d) = d + 2 * next (n4) &
! (next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) &
! (step >= MaxSteps | next(a) > Max | next(b) > Max | next(c) > Max |
   next(d) > Max | next(x) > Max | next(y) > Max | next(n1) > Sup |
   next(n2) > Sup | next(n3) > Sup | next(n4) > Sup )
```

The entire text providing details about the other transitions in the Kripke structure is available from the Appendix.

3.2 Asynchronous Transformation-Communication P Systems Mapped to NuSMV Specifications

We will illustrate the approach using the previous example, Π_1, but we will denote it by Π_2 as it runs in a different way. In the case of asynchronous behaviour the Kripke structure is slightly different. Although it is still possible to use the same states, the transitions will be different, as they reflect the asynchronous behaviour. The above mentioned NuSMV condition expressing maximal parallelism, becomes now

```
next(n1) + next(n2) + next(n3) + next(n4) > 0
```

showing that at least one rule is applied. The entire set of conditions for the *Running* state as well as equations defining the transition from this state to itself are listed below.

```
state = running & next(state) = running &
( next(n1) + next(n2) + next(n3) + next(n4) > 0)  &
( 0 <= next(n1) & next(n1) <= x &
  0 <= next(n2) & next(n2) <= y &
  0 <= next(n3) & next(n3) <= a &
  0 <= next(n4) & next(n4) <= b) &
next(a) = a - next(n3) + next(n1) &
next(b) = b - next(n4) + next(n2) &
next(c) = c + next(n3) &
next(d) = d + 2*next(n4) &
next(x) = x - next(n1) + next(n3) &
next(y) = y - next(n2) + next(n4) &
! (next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0  )&
! (step >= MaxSteps | next(a) > Max | next(b) > Max | next(c) > Max |
   next(d) > Max | next(x) > Max | next(y) > Max | next(n1) > Sup |
   next(n2) > Sup | next(n3) > Sup | next(n4) > Sup )
```

3.3 P Systems with Electrical Charges Mapped to NuSMV Specifications

In the case of electrical charges the above mentioned Kripke structures associated with P systems need to be extended to cope with additional conditions required in this case.

We will consider the following example of a P system with electrical charges and two compartments. $\Pi_3 = (V_3, [[]_2]_1, w_1, w_2, R)$, where $V_3 = \{a, b, c, d, x, y\}$, $w_1 = xy$, $w_2 = \lambda$, $R = \{r_1 : x[]^0_2 \rightarrow [a]^+_2, r_2 : y[]^0_2 \rightarrow [b]^+_2, r_3 : [a \rightarrow xc]^+_2, r_4 : [b \rightarrow ydd]^+_2, r_5 : [x]^+_2 \rightarrow []^0_2 x, r_6 : [y]^+_2 \rightarrow []^0_2 y\}$. The maximal parallelism strategy will be applied in running the system.

The Kripke structure associated with this example will codify the behaviour of the two-compartment P system by using a mapping of the symbols of the alphabet into compartments.

The main change to the Kripke structure represented in NuSMV for transformation-communication P systems consists in adding a new set of conditions, that represent the restrictions imposed by electrical charges associated with compartments. These constraints allow only for some rules to be applied. Two excerpts of NuSMV text for the P system Π_3 are listed below. The first one shows that the second membrane polarisation will become positive after applying rules like $r_1, \ldots r_4$ and neutral in case at least one of the rules r_5, r_6 is applied. Otherwise, if no rule is applied, the polarisation does not change. The second code excerpt shows how the restrictions are applied, e.g. each rule can be applied if the membrane charge is appropriate.

```
next(charge_2) := case
    next(n1) > 0 | next(n2) > 0 | next(n3) > 0 | next(n4) > 0  : 1;
    next(n5) > 0 | next(n6) > 0 : 0;
    1 : charge_2;
esac;

state = running & next(state) = running &
(( charge_2 = 0 & (x_1 > 0 | y_1 > 0 ) &
     x_1 - next(n1) = 0 & y_1 - next(n2) = 0 &
     next(n3) = 0 & next(n4) = 0 & next(n5) = 0 & next(n6) = 0 ) |
 ( charge_2 = 1 & (a_2 > 0 | b_2 > 0 ) &
     a_2 - next(n3) = 0 & b_2 - next(n4) = 0 &
     next(n1) = 0 & next(n2) = 0 & next(n5) = 0 & next(n6) = 0) |
 ( charge_2 = 1 & (x_2 > 0 | y_2 > 0 ) &
     x_2 - next(n5) = 0 & y_2 - next(n6) = 0 &
     next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) ) &
next(x_1) = x_1 - next(n1) + next(n5) &
next(y_1) = y_1 - next(n2) + next(n6) &
next(a_2) = a_2 - next(n3) + next(n1) &
next(b_2) = b_2 - next(n4) + next(n2) &
next(c_2) = c_2 + next(n3) &
next(d_2) = d_2 + 2 * next(n4) &
next(x_2) = x_2 - next(n5) + next(n3) &
next(y_2) = y_2 - next(n6) + next(n4) &
! ( next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0 &
    next(n5) = 0 & next(n6) = 0 ) &
! ( step >= MaxSteps | next(x_1) > Max | next(y_1) > Max |
    next(a_2) > Max | next(b_2) > Max | next(c_2) > Max |
    next(d_2) > Max | next(x_2) > Max | next(y_2) > Max |
    next(n1) > Sup | next(n2) > Sup | next(n3) > Sup |
    next(n4) > Sup )
```

4 Formal Verification Using NuSMV

In this section we will show how a P system mapped into NuSMV is verified for certain properties, by using model checking techniques. The properties that will be checked are first identified by Daikon, a tool which dynamically detects program invariants based on execution traces. Following the strategy exposed in [2], we synthesise the traces from the P-Lingua environment's execution data[8], and then run Daikon to generate an extended list of invariants, which help us formulate the LTL properties. The Daikon tool is even able to detect some mathematical relationships between various variables of the system, based on complex mathematical functions, not all of them expressible in NuSMV. Both, the translation from P-Lingua specification to NuSMV specification and the P-Lingua traces conversion to Daikon inputs are obtained in an automatic way.

We will refer to the three examples presented above and to a nondeterministic variant of the predator-prey problem [10] in order to illustrate what kind of properties we can check. There are various ways to classify the properties we aim to verify. In certain areas there have been identified specific types of queries categorised as patterns [16]. We will refer to some of these in the presentation below. We will present those properties as they have been captured by the Daikon tool and as LTL expressions.

We have initially looked at those properties that state the main invariants of the system. These invariants represent one of the main sets of patterns in [16]. It is obvious that we have been after properties like "two times the number of c's equals the number of d's" in the examples given by Π_1 and Π_3. Indeed, these properties have been identified by the Daikon tool as

```
2 * c - d == 0
```

or

```
2 * orig(c) - orig(d) == 0
```

where `orig(c)` means `c` in the previous step.

This property is then checked with the LTL query `G ( 2 * c - d = 0)`. It is easy to observe that this invariant does not hold for the P system Π_2, which works in an asynchronous way. Indeed this is neither returned by Daikon nor verified by NuSMV. This is a good example of a property that returns a test sequence for our system.

Other properties that are extracted by Daikon from the traces generated by P-Lingua describe some expected properties of a correct model. For instance in all these examples we expect that the number of occurrences of each of the variables `a, b, x, y` is either 0 or 1. This is present in the list of properties identified by Daikon and the NuSMV model checker shows this is true (in Daikon they occur as `a one of { 0, 1 }` or `b is boolean` and in NuSMV are expressed as LTL expressions `G ( 0 <= a & a <= 1 )` or `G ( 0 <= b & b <= 1 )`).

Some properties extracted by Daikon reveal relationships between elements of the multiset across development, sometimes involving different steps. These are not always obvious and can be utilised to generate some further more complex conditions. For example for the first P system, Π_1, it is identified the property `2 * c - 2 * orig(a) == orig(d)` which links `c` with `a, d` occurring in the previous step. This property holds for this example as it is shown by NuSMV.

Daikon was able to identify simple forms of consequence patterns [16], when the two states appear one after the other. For the Π_1 P system a relationship between consecutive occurrences of c is stated as `(c == 0) ==> (orig(c) == 0)`, which is true as it is shown by the NuSMV formula `G((c=0)->(c_old=0))`.

The nondeterministic variant of the predator-prey problem can be defined by the following P system, $\Pi_{PP} = (V, [\,]_1, w_1, R_1)$, where $V = \{a, x, y, b\}$, $R_1 = \{ax \rightarrow xx, xy \rightarrow yy, y \rightarrow b\}$ and w_1 is the initial multiset. We considered simulations for $w_1 = a^{100}x^{100}y^{10}$ and simulations and verifications using NuSMV for $w_1 = a^{10}x^{10}y^5$. This system simulates the interplay between preys, x's, and predators, y's, in an environment with a fixed amount of resources, a's. Preys breed when resources are available and are eaten by predators which also die – the last rule. Various simulations and analysis made with Daikon reveal consistently some invariants of this problem; of them only `(b == 0) ==> (orig(b) == 0)` and `(orig(a) == 0) ==> (a == 0)` are validated by NuSMV (`G ( (step > 1 & b = 0) -> b_old = 0)` and `G ( (step > 1 & a_old = 0) -> a = 0)`, respectively). These show that if at a moment in time the number of death predators is 0 then this is true for all the previous steps and if the resources are exhausted they will remain the same. Other potential invariants, like `(a <= x)` or `(x > y)` or `(b < x)`, are not true and NuSMV confirms this. They can be true only for some executions, but not in general, irrespective of the initial values.

A comprehensive list of Daikon invariants and associated LTL specifications for the above examples has been collected and is available from EvoMT website `http://fmi.upit.ro/evomt/psys/psys_daikon.html`.

5 Conclusions

This paper has investigated a methodology to verify P systems specifications, by first identifying these properties as invariants produced by the Daikon tool and then formally checking whether these are true, by using NuSMV. The benefits of this methods have been identified and assessed through some case studies.

The methods suffers from the well known scalability issues most of the model checking based approaches exhibit.

In our future research we aim to overcome some of the limitations of the above presented approach, by better codifications of the systems and better formulated properties to be checked. We also intend to link the formal verification with testing and to expand it to other classes of P systems, including stochastic P systems which start to be increasingly used in various applications.

Acknowledgements

This research of MG, FI and RL was supported by CNCSIS - UEFISCSU, project number PNII - IDEI 643/2008. The authors would like to thank the anonymous reviewers for their useful comments and for suggesting the use of the prey-predator example to illustrate the interplay between various tools employed.

References

1. Andrei, O., Ciobanu, G., Lucanu, D.: A rewriting logic framework for operational semantics of membrane systems. Theor. Comput. Sci. 373(3), 163–181 (2007)
2. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2007. LNCS, vol. 4860, pp. 138–159. Springer, Heidelberg (2007)
3. Ciobanu, G.: Semantics of P systems. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) Handbook of Membrane Computing, ch. 16, pp. 413–436. Oxford University Press, Oxford (2010)
4. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Natural Computing Series. Springer, Heidelberg (2006)

5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
6. Clarke Jr, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
7. Dang, Z., Ibarra, O.H., Li, C., Xie, G.: On the decidability of model-checking for P systems. Journal of Automata, Languages and Combinatorics 11(3), 279–298 (2006)
8. Díaz-Pernil, D., Graciani, C., Gutiérrez-Naranjo, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Software for P systems. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) Handbook of membrane computing, ch. 17, pp. 437–454. Oxford University Press, Oxford (2010)
9. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC 1982, pp. 169–180. ACM Press, New York (1982)
10. Fontana, F., Manca, V.: Predator-prey dynamics in P systems ruled by metabolic algorithm. Biosystems 91, 545–557 (2008)
11. Gheorghe, M., Ipate, F.: On testing P systems. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 204–216. Springer, Heidelberg (2009)
12. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
13. Ipate, F., Gheorghe, M.: Testing non-deterministic stream X-machine models and P systems. Electronic Notes in Theoretical Computer Science 227, 113–126 (2009)
14. Ipate, F., Gheorghe, M., Lefticaru, R.: Test generation from P systems using model checking. Journal of Logic and Algebraic Programming 79(6), 350–362 (2010)
15. Kleijn, J., Koutny, M.: Petri nets and membrane computing. In: Păun, G., Rozenberg, G., Salomaa, A. (eds.) Handbook of membrane computing, ch. 15, pp. 389–412. Oxford University Press, Oxford (2010)
16. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. Bioinformatics 24(16), 227–233 (2008)
17. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, FOCS 1977, pp. 46–57. IEEE, Los Alamitos (1977)
18. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science 13, 45–60 (1981)
19. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
20. Păun, G.: Membrane Computing: An Introduction. Springer, Heidelberg (2002)
21. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford (2010)
22. Şerbănuţă, T., Ştefănescu, G., Roşu, G.: Defining and executing P systems with structured data in K. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)

Appendix

NuSMV Specification

Based on the P system specification, the tool developed by the authors generates a SMV file, that will be processed by the NuSMV model checker.

```
-- This is a 1-membrane P system working in a maximally parallel manner
-- The NuSMV file is automatically generated
-- The P system consists of:
-- Alphabet = [a, b, c, d, x, y]
-- Initial multiset = x, y
-- Rules:
-- r1 : x --> a
-- r2 : y --> b
-- r3 : a --> x, c
-- r4 : b --> y, d*2

MODULE main

VAR
   a : 0..15;
   b : 0..15;
   c : 0..15;
   d : 0..15;
   x : 0..15;
   y : 0..15;
   a_old : 0..15;
   b_old : 0..15;
   c_old : 0..15;
   d_old : 0..15;
   x_old : 0..15;
   y_old : 0..15;
   n1 : 0..15;
   n2 : 0..15;
   n3 : 0..15;
   n4 : 0..15;
   state : {running, halt, crash};
   step : 0..15;

DEFINE
   Max := 10;
   MaxSteps := 10;
   Sup := 10;

ASSIGN
   init(a) := 0;
   init(b) := 0;
   init(c) := 0;
   init(d) := 0;
   init(x) := 1;
   init(y) := 1;
   init(a_old) := 0;
   init(b_old) := 0;
   init(c_old) := 0;
   init(d_old) := 0;
   init(x_old) := 0;
   init(y_old) := 0;
   init(n1) := 0;
   init(n2) := 0;
   init(n3) := 0;
   init(n4) := 0;
   init(state) := running;
   init(step) := 0;

ASSIGN
   next(a_old) := a;
   next(b_old) := b;
   next(c_old) := c;
   next(d_old) := d;
   next(x_old) := x;
   next(y_old) := y;

   next(step) := case
      (step <= MaxSteps & state=running) :  step + 1;
      1 : step;
   esac;

TRANS

-- STATE = running
state = running & next(state) = running &
x - next(n1) = 0 & y - next(n2) = 0 & a - next(n3) = 0 & b-next(n4) = 0 &
next(a) = a - next(n3) + next(n1) &
next(b) = b - next(n4) + next(n2) &
next(c) = c + next(n3) &
next(d) = d + 2*next(n4) &
next(x) = x - next(n1) + next(n3) &
next(y) = y - next(n2) + next(n4) &
! (next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0  )&
! (step >= MaxSteps | next(a) > Max | next(b) > Max | next(c) > Max |
   next(d) > Max | next(x) > Max | next(y) > Max | next(n1) > Sup |
   next(n2) > Sup  | next(n3) > Sup | next(n4) > Sup ) |

state = running & next(state) = halt &
x - next(n1) = 0 & y - next(n2) = 0 & a - next(n3) = 0 & b-next(n4) = 0 &
next(a) = a - next(n3) + next(n1) &
next(b) = b - next(n4) + next(n2) &
next(c) = c + next(n3) &
next(d) = d + 2*next(n4) &
next(x) = x - next(n1) + next(n3) &
next(y) = y - next(n2) + next(n4) &
(next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) &
! (step >= MaxSteps | next(a) > Max | next(b) > Max | next(c) > Max |
   next(d) > Max | next(x) > Max | next(y) > Max | next(n1) > Sup |
   next(n2) > Sup | next(n3) > Sup | next(n4) > Sup) |

state = running & next(state) = crash &
x - next(n1) = 0 & y - next(n2) = 0 & a - next(n3) = 0 & b-next(n4) = 0 &
next(a) = a - next(n3) + next(n1) &
next(b) = b - next(n4) + next(n2) &
next(c) = c + next(n3) &
next(d) = d + 2*next(n4) &
next(x) = x - next(n1) + next(n3) &
next(y) = y - next(n2) + next(n4) &
! (next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0) &
  (step >= MaxSteps | next(a) > Max | next(b) > Max | next(c) > Max |
   next(d) > Max | next(x) > Max | next(y) > Max | next(n1) > Sup |
   next(n2) > Sup | next(n3) > Sup | next(n4) > Sup) |

-- STATE = HALT
state = halt & next(state) = halt &
next(n1) = 0 & next(n2) = 0 & next(n3) = 0 & next(n4) = 0 &
next(a) = a & next(b) = b & next(c) = c & next(d) = d & next(x) = x &
 next(y) = y |

-- STATE = CRASH
state = crash & next(state) = crash &
next(n1) = n1 & next(n2) = n2 & next(n3) = n3 & next(n4) = n4 &
next(a) = a & next(b) = b & next(c) = c & next(d) = d & next(x) = x &
 next(y) = y

-- Simple LTL checks
LTLSPEC   G ( a = b )
LTLSPEC   G ( x = y )
LTLSPEC   G ( 0 <= a & a <= 1 )
LTLSPEC   G ( d mod 2 = 0)
LTLSPEC   G ( 0 <= x & x <= 1 )
LTLSPEC   G ( step > 1 & state = running -> a = x_old )
LTLSPEC   G ( 2 * c - d = 0)
```