View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by idUS. Depósito de Investigación Universidad de Sevilla

Towards an Integrated Approach for Model Simulation, Property Extraction and Verification of P Systems

Raluca Lefticaru[1], Florentin Ipate[1], Luis Valencia Cabrera[2],
Adrian Ţurcanu[1], Cristina Tudose[1], Marian Gheorghe[3],
Mario de J. Pérez-Jiménez[2], Ionuţ Mihai Niculescu[1], and Ciprian Dragomir[3]

[1]Department of Mathematics and Computer Science, University of Piteşti
Str. Târgu din Vale 1, 110040, Piteşti, Romania
`{name.surname}@upit.ro`

[2]Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012, Sevilla, Spain
`{lvalencia,marper}@us.es`

[3]Department of Computer Science, University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
`{m.gheorghe,c.dragomir}@sheffield.ac.uk`

Summary. This paper presents an integrated approach for model simulation, property extraction and formal verification of P systems, illustrated on a tissue P system with active membranes solving the 3-colouring problem. The paper focuses on this problem and reports the invariants and the properties extracted and verified using a series of tools (Daikon, MeCoSim, Maple, Spin, ProB) and languages (P–Lingua, Promela, Event-B). Appropriate tools and integration plugins, which facilitate and even automate the steps involved in the aforementioned approach, have also been developed. The case study chosen is complex (it involves an exponential growth of the number of states through the use of membrane division rules) and the properties obtained are non-trivial.

1 Introduction

Inspired by the behaviour and structure of the living cell, P systems have emerged in recent years as powerful computational tools [21]. Many variants of P systems have been introduced and a number of theoretical aspects have been intensely studied: the computational power of different variants, their capabilities to solve hard problems, like NP-complete ones, decidability, complexity aspects and hierarchies of classes of languages produced by these devices [23]. In the last years there have also been significant developments in using the P systems paradigm to model, simulate and formally verify various systems [5, 23].

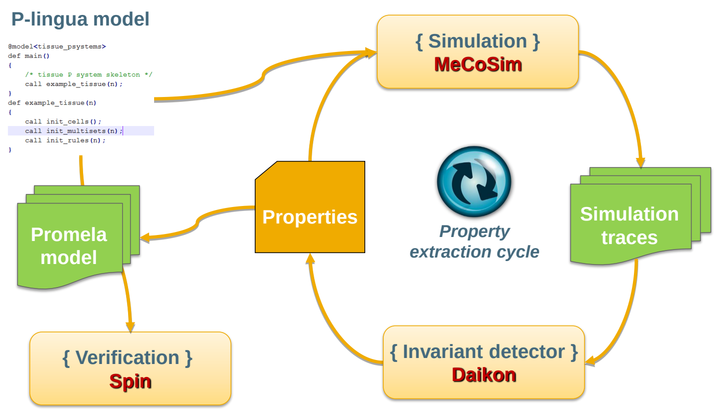
Up to now, two main areas concerning P system formal verification have been investigated: property verification through model checking and property extraction. However, there is no approach which integrates these two aspects (or other related aspects such as simulation). Initial research on P system model checking has tackled the problem of identifying decidable (or undecidable) problems [7, 8]. Verifying properties for P systems implies defining and implementing an operational semantics of the P system and using a corresponding model-checker. Among the tools used we mention: Maude [2], the probabilistic model-checker Prism [24, 4], the symbolic model verifier NuSMV [17], the Spin [19, 18] and ProB model checkers [16].

Property extraction using Daikon, a dynamic invariant detector, and further verification of the P systems was tackled in [4, 15]. In [4] a simple example involving a regulatory network is presented, along with the properties (preconditions, postconditions, invariants) inferred by Daikon. The relationships discovered regard the boundaries of the number of objects (e.g., $0 \leq prot \leq 205$) and relations between objects, such as $rna < orig(rep)$ or $(rna = 0) \rightarrow (prot = 0)$. They have been checked using the Prism probabilistic model checker. In [15] simple cell like P systems have been used and invariants like $2 * c - d = 0$, $(b = 0) \rightarrow (orig(b) = 0)$ have been obtained and further verified using NuSMV.

In this paper, we propose an integrated methodology for modelling, simulation, analysis, property extraction (invariant detection) and verification through model checking for P systems. The approach integrates a modelling and simulation environment (P–Lingua and MeCoSim) with model checkers, property extraction tools (Daikon) and tools for mathematical and symbolic calculus (Maple). Appropriate integration tools (plugins) have also been developed (see Fig. 1).

Starting from a problem, this process involves: the modelling of the problem by means of P systems, the model transcription into a language like P–Lingua [14], understandable by a machine; the definition of a visual interface, to enter the needed inputs and show the desired outputs from the computation; the simulation of the model under different initial parameters; the data extraction from the simulation; the invariants detection from the extracted data, and the analysis and verification of the detected properties. A detailed description of the methodology has been provided in section 4, applying the process until the invariants detection for the 3–Col problem in subsection 4.4. The approach is illustrated on a case study involving the tissue P system model for the well-known 3-colouring (3-Col) problem [10, 12].

The paper is structured as follows. We start by presenting in Section 2 the notation and main concepts to be used in the paper. Section 3 presents a set of initial properties for the 3-colouring problem, which have been verified using Spin and ProB model checker. In the next two sections are presented a methodology for properties extraction (invariants detection), its integration with the MeCoSim platform [20] and the empirical detection and validation of additional properties. Finally, conclusions are drawn in Section 6.



```
@model<tissue_psystems>
def main()
{
    /* tissue P system skeleton */
    call example_tissue(n);
}
def example_tissue(n)
{
    call init_cells();
    call init_multisets(n);
    call init_rules(n);
}
```

Fig. 1. Methodology Overview

2 Background

Before presenting our approach, let us establish the notations used and define the class of P systems addressed in the paper.

Given a finite alphabet $V = \{a_1, ..., a_p\}$, a *multiset* is either denoted by a string $u \in V^*$ (in which the order is not important, the string notation is only used as a convention), or by an associated vector of non-negative integers, $\Psi_V(u) = (|u|_{a_1}, ..., |u|_{a_p})$, where $|u|_{a_i}$ denotes the number of a_i occurrences in u, for each $1 \leq i \leq p$.

The following definition refers to a model of tissue P systems with cell division, introduced in [22]. This model can be seen as a network of cells, whose structure is not static: it is inspired by the way cells are duplicated in a natural way via mitosis.

Definition 1. *Formally, a* tissue P system with cell division *of degree* $q \geq 1$ *is a tuple of the form*

$$\Pi = (\Gamma, w_1, \ldots, w_q, \varepsilon, R, i_0),$$

where:

1. $q \geq 1$ *is the initial* degree *of the system; the system contains* q *cells, labelled with* $1, 2, \ldots, m$*;* 0 *represents the environment.*

2. *Γ is a finite alphabet (called working alphabet), whose symbols will be called objects.*
3. *$w_1, \ldots, w_q$ are strings over Γ, describing the multisets of objects placed in the q cells of the system.*
4. *$\varepsilon \subseteq \Gamma$ is the set of objects present in the environment in arbitrarily many copies each.*
5. *R is a finite set of* developmental rules *of the following form:*
   a) Communication rules: *$(i, u/v, j)$, for $i, j \in \{0, 1, 2, \ldots, q\}$, $i \neq j$, $u, v \in \Gamma^*$. When applying a rule $(i, u/v, j)$, the objects of the multiset represented by u are sent from region i to region j and simultaneously the objects of the multiset v are sent from region j to region i.*
   b) Division rules: *$[a]_i \to [b]_i[c]_i$, where $i \in \{1, 2, \ldots, q\}$ and $a, b, c \in \Gamma$. The cell with label i is divided in two cells with the same label; in the first copy the object a is replaced by b, in the second copy the object a is replaced by c; all other objects are replicated and copies of them are placed in the two new cells.*
6. *$i_0 \in \{0, 1, 2, \ldots, q\}$ denotes the* output region *(which can be the region inside a membrane or the environment).*

Rules are applied as usual in a maximally parallel way, with only one restriction: when a cell is divided, the division rule is the only one which is applied for that cell in that step; the objects inside that cell do not evolve in that step.

This class of P systems can be further extended, for solving NP-complete problems, to recognizer P systems. A *recognizer tissue P system with cell division* is a tuple $(\Gamma, \Sigma, w_1, \ldots, w_q, \varepsilon, R, i_{in}, i_0)$, which has, in addition to a tissue P system with cell division:

- Two distinguished objects $yes, no \in \Gamma$, present in at least one copy in $w_1, w_2, \ldots, w_q$, but not present in ε.
- An input alphabet Σ strictly contained in Γ.
- An input cell $i_{in} \in \{1, \ldots, q\}$.

Also, it must satisfy the followings:

- The output region i_0 is the environment.
- All computations halt.
- If C is a computation of Π, then either the object *yes* or the object *no* (but not both) must have been released into the environment, and only in the last step of the computation.

3 Verifying a first set of properties for the 3-colouring problem

In this section we will introduce a simplified version of a tissue P system solving the 3-colouring problem, we will present some of its properties and their verification using the Spin and ProB model checkers.

3.1 A P system for the 3-colouring problem

In order to illustrate our approach regarding property extraction and verification for P systems, we have considered a simplified version of the 3-colouring problem from [9, 11].

The k-colouring problem is formulated as follows: given an undirected graph $G = (V, E)$, decide whether or not G is k-colourable; that is, if there exists a valid k-colouring of G (for every edge $\{u, v\} \in E$ the colours of u and v are different).

The 3-colouring problem can be solved in linear time by a family of recognizer tissue P systems with cell division [9]. The solution proposed in [9] is using a brute force algorithm, in the framework of recognizer tissue P systems with cell division, which consists of 4 stages:

1. Generation Stage: an initial cell, labelled by 2, is divided into two new cells; this process is repeated until all possible candidate solutions to the problem are generated (one solution for each membrane).
2. Prechecking Stage: after obtaining all possible 3-colourings (in cells labelled by 2), additional objects are generated in the cells, for every edge of the graph.
3. Checking Stage: it is verified if there exists a pair of adjacent vertices in the graph, with the same colour in the corresponding candidate solution.
4. Output Stage: the system sends to the environment the right answer according to the results of the previous stage (*yes* or *no*).

As we will focus in the rest of the paper only on the properties from the Generation Stage, we will omit from the recognizer P system model given in [9] some rules and objects, which would hinder the understanding of the mechanism. More precisely, we will consider only the division rules and a restricted set of objects, so we can define the model using the basic class of tissue P systems with cell division. However, for a complete specification in terms of a family of recognizer tissue P systems, [9] can be consulted.

Let $\Pi(n) = (\Gamma(n), w_1, w_2(n), \varepsilon, R(n), i_0)$ be a family of tissue P systems with cell division of degree 2, where:

1. $\Gamma(n) = \{A_i, R_i, T_i, B_i, G_i : 1 \leq i \leq n\}$
2. $w_1 = \emptyset$, $w_2(n) = \{A_1, \ldots, A_n\}$
3. $R(n)$ is a set of division rules:
   - $r_{1,i} \equiv [A_i]_2 \rightarrow [R_i]_2[T_i]_2$ for $i = 1, \ldots, n$
   - $r_{2,i} \equiv [T_i]_2 \rightarrow [B_i]_2[G_i]_2$ for $i = 1, \ldots, n$

In this model A_i encodes the i-th vertex of the graph; R_i, B_i, G_i represent the three colours red, blue, green. Appendix A presents two examples of computation for $\Pi(2)$ and $\Pi(3)$. It can be observed that, after appropriate divisions, in the step $2n$ we get exactly 3^n cells encoding all the possible 3-colourings of the graph having vertices $A_1, \ldots, A_n$. Appendix B presents the number of cells labelled 2 at each computation step for $2 \leq n \leq 11$, simulation results which help us formulate some interesting properties.

Given the family of P systems $\Pi(n)$ previously defined, an initial set of properties have been identified manually (without using property extraction tools):

P_1 For each computation C of the P system $\Pi(n)$, there are 3^n cells labelled with 2 at configuration C_{2n}.

P_2 For each computation C of the P system $\Pi(n)$, the configuration C_{n+1} has exactly $2^{n+1}-1$ cells labelled 2.

P_3 For each computation C of the P system $\Pi(n)$, for each $0 \leq j \leq n$ the configuration C_j has exactly 2^j cells labelled 2.

P_4 For each combination $(X_1, X_2, \ldots, X_n)$, $X_i \in \{R_i, G_i, B_i\}$, $i = 1 \ldots n$, there exists, at configuration C_{2n}, one and only one cell labelled by 2 that contains the multiset $\{X_1, X_2, \ldots, X_n\}$.

3.2 3-Col property verification using model checking

This initial set of properties is now verified using two model checkers, Spin and ProB.

Spin is a model checker widely used in industries that build critical systems and is considered one of the most powerful model checkers available [3]. It is designed for modelling and verifying concurrent and distributed systems specified in Promela (Process or Protocol Meta Language), a verification modelling language. The properties to be verified can be expressed in LTL or by using assertion statements.

ProB is an animation and model checking tool integrated within the Rodin platform, which accepts Event-B models [1]. Unlike most model checking tools, ProB works on higher-level formalisms and so it enables a more convenient modelling. Besides verification of properties (expressed using the LTL or the CTL formalism), it also provides animation facilities, allowing to visualize, at any moment, the state space or to execute a given number of operations.

Property verification using Spin

As explained in [18], the executable specification for the Spin model checker associated to a P system will contain extra states and variables, corresponding to intermediate steps, which have no correspondence in the P system configurations. For this reason, the properties to be verified, that refer to the P system, need to be reformulated as equivalent LTL formulas for the associated Promela implementation.

The P system properties verified in this section are of the form 'G $(\phi \rightarrow \psi)$', and the equivalent LTL formula for the Promela model is '$[](!\phi \; ||\psi|| \; !pInS)$', as formally proven in [19] ($pInS$ is used to express if the current configuration in the Promela model represents also a state in the P system).

In our experiments, we have used one Promela specification file for each particular P system $\Pi(n)$, for all $n \in \{2, \ldots, 9\}$ because these files were (semi-)

automatically generated from each corresponding instance of P–Lingua definition file using the `plinguacore` library. We have successfully simulated all these models with Spin, for $n \in \{2, \dots, 9\}$; typical state explosion problems appeared when we attempted to verify the properties mentioned earlier for $n \geq 4$.

For $n \in \{2, 3\}$ we have verified all the formulas presented bellow, using also some techniques that Spin provides to reduce the memory use. Starting with $n = 4$, we obtained `out of memory` for properties that involved checking many steps of the computations.

The properties verified, expressed as LTL formulas for the Spin model checker, for $n = 3$, are:

P_1 :[] $((!(noOfSteps == 6) \mid\mid (noOfCells == 27) \mid\mid (!pInS))$
P_2 :[] $((!(noOfSteps == 4) \mid\mid (noOfCells == 15) \mid\mid (!pInS))$
P_3 :[] $((!(noOfSteps >= 0 \,\&\&\, noOfSteps <= 3)) \mid\mid (noOfCells == pow2noOfSteps) \mid\mid (!pInS))$, where $pow2noOfSteps$ is a variable which computes $2^{noOfSteps}$
P_4 :This property is hard to verify with Spin because of the complex operations involved.

Property verification using ProB

The Event-B model of a P system with active membranes can be specified using two functions *cell* and *cellp*, representing the number of objects of each type contained in every cell and the number of objects produced between two steps of maximal parallelism, respectively. The rules are represented by events. Each division rule adds a cell to the domain of these functions. Additionally, a special event called *update*, enabled after each step of maximal parallelism, is used to add each value of *cellp* to *cell* and to reset all the values of *cellp* to 0. More details can be found in [6].

First, we developed an Event-B model, using the Rodin platform, for each particular P system $\Pi(n)$, with $n \in \{2, 3, 4\}$. Then, the possibility to use quantifiers in ProB allowed us to develop a general Event-B model for the family of P systems $\Pi(n)$, that has been instantiated for particular values of n. We animated the models in order to see how the system evolves and we verified their properties using the model checker ProB. Unlike the Promela specification, where the number of cells with label 2 was incremented after each division rule, we could specify the properties P_1-P_4 neglecting the extra intermediary) states; this is because these properties refer only to the number of steps of maximal parallelism in the evolution of $\Pi(n)$ (counted by a variable called *noOfSteps*) and to the number of cells with label two (counted by a variable called *noOfCells*), whose values are modified only in the *update* event. On the other hand, we used an additional state, *Halt*, to mark final configurations. However, we were able to verify all the properties only for $n \in \{2, 3, 4\}$; for $n = 5$, due to the state explosion problem, the model checker crashed with an out of memory error before reaching the final configuration (after producing 196 cells labelled 2). Consequently, for $n > 5$, we could verify only some simple properties, that do not involve terminal configuration (e.g. P_3 for small values of j).

The properties were specified using LTL as follows:

P_1 :$G\{state = Halt \Rightarrow noOfSteps = 2 * n \,\& noOfCells = 3^n\}$
P_2 :$G\{noOfSteps = n + 1 \Rightarrow noOfCells = 2^{n+1} - 1\}$
P_3 :$G\{!j.j >= 0 \,\& j <= n \,\&\, noOfSteps = j \Rightarrow noOfCells = 2^j\}$
P_4 :We were able to verify P_4 splitting it in two properties, the first one for the existence and the second one for the unicity. For $n = 2$ these properties were formulated as follows:

– For all x, y symbols in $\{R_i, G_i, B_i\}$, $i \in \{1, 2\}$, there exist one cell in the final configuration that contains one x and one y:
$G\{state = Halt \Rightarrow (!x, y.x : \{R_1, R_2, G_1, G_2, B_1, B_2\} \,\&\, y : \{R_1, R_2, G_1, G_2, B_1, B_2\} \,\&\, (x/ = y) \Rightarrow (\#c.c : dom(cell) \,\&\, cell(c)(x) = 1 \,\&\, cell(c)(y) = 1))\}$

– In the final configuration any two different cells $c1$, $c2$ have different contents:
$G\{state = Halt \Rightarrow (!c1, c2.c1 : dom(cell) \,\&\, c2 : dom(cell) \,\&\, (c1/ = c2) \Rightarrow (\#s.s : \{R_1, R_2, G_1, G_2, B_1, B_2\} \,\&\, cell(c1)(s)/ = cell(c2)(s)))\}$
For higher values of n the formulas for these properties are very large and for space considerations, we will omit them.

Here "!", "#" and ":" correspond to the universal quantifier "$\forall$", existential quantifier "$\exists$" and membership operator "$\in$", respectively.

4 Integrating Daikon in MeCoSim and finding new relations

The next stage in the proposed methodology is the automatic extraction of new properties from simulation traces. The main tools used (MeCoSim and Daikon) and their integration are presented next. The process of identifying new properties, broken down in a number of individual steps, is also described.

4.1 Modelling and formalization

Once the 3-Col problem has been studied and modelled by means of P systems, this model must be expressed in a language that may be understood by a simulation machine. For this purpose, the standard P–Lingua [14] language has been chosen as our modelling language. The initial multisets and rules are expressed as follows:

```
@ms(2) += A{i} : 1<=i<=n;

/* r1 */ [A{i}]'2 --> [R{i}]'2 [T{i}]'2 : 1<=i<=n;
/* r2 */ [T{i}]'2 --> [B{i}]'2 [G{i}]'2 : 1<=i<=n;
```

The full code of the model in P–Lingua is showed in Appendix C. This file will serve as an input for MeCoSim, so that the P system can be simulated, analyzed and debugged, and invariants can be detected, to be then verified by model checking.

4.2 MeCoSim

In order to provide an integrated methodology for model simulation, properties extraction and verification, we need an integrated environment to simplify the user's process.

In this sense, a general purpose membrane computing simulator, **MeCoSim** [20], was provided. It was initially designed to enable the user defined customized interfaces, with inputs, outputs, charts, etc., adapted to each family of P systems. This permits entering data for different initial conditions, instantiating different P systems of the family.

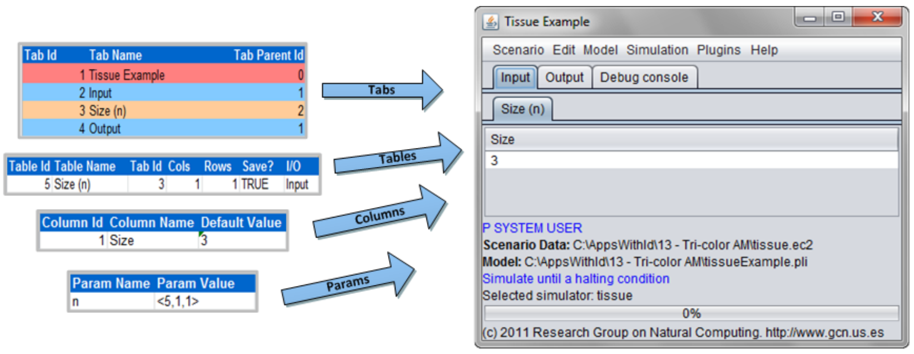
The initial aim of this software environment has been extended such that it can cover a more general set of applications by providing flexible and powerful methods to integrate various software applications and packages as MeCoSim plugins. These kind of plugins can be easily added to MeCoSim by setting appropriate parameters in a configuration file. Keeping in mind this architecture and the developed plugins, MeCoSim may provide a platform for the integration of different tools for the modelling, simulation, analysis, property extraction and verification of P systems. Some of this tools have already been developed and/or integrated, others are being developed, and many other could be added in a similar way.

To take advantage of this framework for studying the 3-Col problem, we need to define our customized inputs, outputs, extractions, etc. The main steps of this process are illustrated in the next paragraphs.

MeCoSim permits setting the hierarchy of tabs to be shown in the visual user interface, including input and output tables inside each leaf tab. In our case, we divide the information in two tabs, Input and Output (plus an additional tab, Debug console, provided by default in MeCoSim, used for debugging the models). For our example, we only need one input parameter, n, so one input table is defined inside the tab Input, as showed in Fig. 2.

Now the simulation could be performed, in such a way that n takes the value from the input table, this parameter complements the P–Lingua file to instantiate the initial configuration of the P system and the computation steps run until a halting condition is reached. In the debug console, we can run step by step, looking at all the objects of the multisets inside each cell and compartment, for each computation step.

However, this process could be very slow if we are interested in several objects, membranes or steps, so we need a way to define customized outputs showing the desired information only. MeCoSim provides this mechanism, and we define outputs as shown in Fig. 3 to study different issues: entire configurations, objects per membrane, objects by type (R, G, B), number of cells, etc.



Tab Id	Tab Name	Tab Parent Id
1	Tissue Example	0
2	Input	1
3	Size (n)	2
4	Output	1

Table Id	Table Name	Tab Id	Cols	Rows	Save?	I/O
5	Size (n)	3	1	1	TRUE	Input

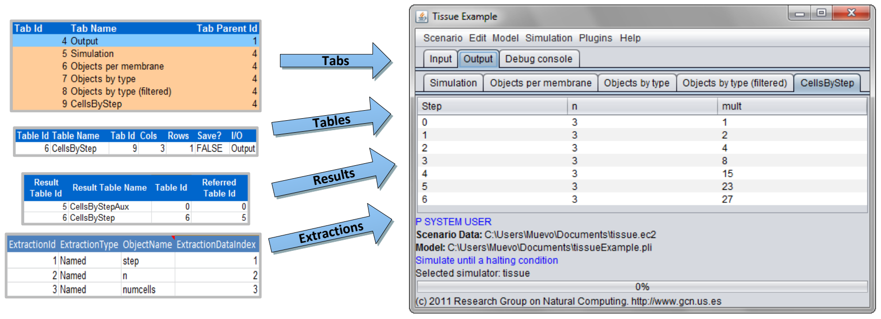
Column Id	Column Name	Default Value
1	Size	3

Param Name	Param Value
n	<5,1,1>

Size
3

P SYSTEM USER
Scenario Data: C:\AppsWithId\13 - Tri-color AM\tissue.ec2
Model: C:\AppsWithId\13 - Tri-color AM\tissueExample.pli
Simulate until a halting condition
Selected simulator: tissue
0%
(c) 2011 Research Group on Natural Computing. http://www.gcn.us.es

Fig. 2. MeCoSim window - Input tab: value of n



Tab Id	Tab Name	Tab Parent Id
4	Output	1
5	Simulation	4
6	Objects per membrane	4
7	Objects by type	4
8	Objects by type (filtered)	4
9	CellsByStep	4

Table Id	Table Name	Tab Id	Cols	Rows	Save?	I/O
6	CellsByStep	9	3	1	FALSE	Output

Result Table Id	Result Table Name	Table Id	Referred Table Id
5	CellsByStepAux	0	0
6	CellsByStep	6	5

ExtractionId	ExtractionType	ObjectName	ExtractionDataIndex
1	Named	step	1
2	Named	n	2
3	Named	numcells	3

Step	n	mult
0	3	1
1	3	2
2	3	4
3	3	8
4	3	15
5	3	23
6	3	27

P SYSTEM USER
Scenario Data: C:\Users\Muevo\Documents\tissue.ec2
Model: C:\Users\Muevo\Documents\tissueExample.pli
Simulate until a halting condition
Selected simulator: tissue
0%
(c) 2011 Research Group on Natural Computing. http://www.gcn.us.es

Fig. 3. MeCoSim window - Output tab: number of cells by step

Eventually, as we see in Fig. 3, we set the information to be extracted for Daikon. This makes up the output files from the simulation, to be used as an input for the MeCoSim Daikon plugin.

4.3 MeCoSim new plugin for Daikon integration

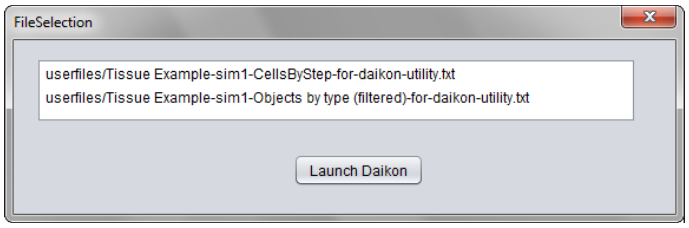
MeCoSim provides an easy way to add plugins, enriching the default functionality. Taking advantage of this architecture, a new plugin has been developed to integrate Daikon with MeCoSim.

Daikon [13] is a tool which dynamically detects programs invariants, based on their execution traces. It can discover properties from C, C++, Eiffel, Java, or Perl programs, from spreadsheet files and other data sources. The usual operation of Daikon is the following: it receives data trace files about the values of some variables across a sequence of steps from the execution of a program, and tries to detect properties of types: precondition, postcondition and invariant. We are mainly interested in the last one, but the previous ones could be also useful for checking the correctness of the models.

As part of the proposed methodology, we aim to integrate this tool with MeCoSim, so invariants could be detected from the desired outputs of the simulation. For this purpose, a plugin has been developed; an overview of the entire (simulation and property extraction) process involves the following steps:

1. The model of the P system (written in P–Lingua, possibly parametrized) is loaded in MeCoSim.
2. The initial parameters (n in the case of the 3–Col problem) for instantiating the specific P system are provided by the user in a visual way through the input tables.
3. The simulation runs, generating extraction files for the outputs previously set.
4. The plugin can be called from a menu option (Plugins > "Daikon").

When the plugin is launched from MeCoSim, a window with a listing of available extraction files is visualized, as showed in figure 4.



userfiles/Tissue Example-sim1-CellsByStep-for-daikon-utility.txt
userfiles/Tissue Example-sim1-Objects by type (filtered)-for-daikon-utility.txt

Fig. 4. MeCoSim window - Daikon plugin - File selection

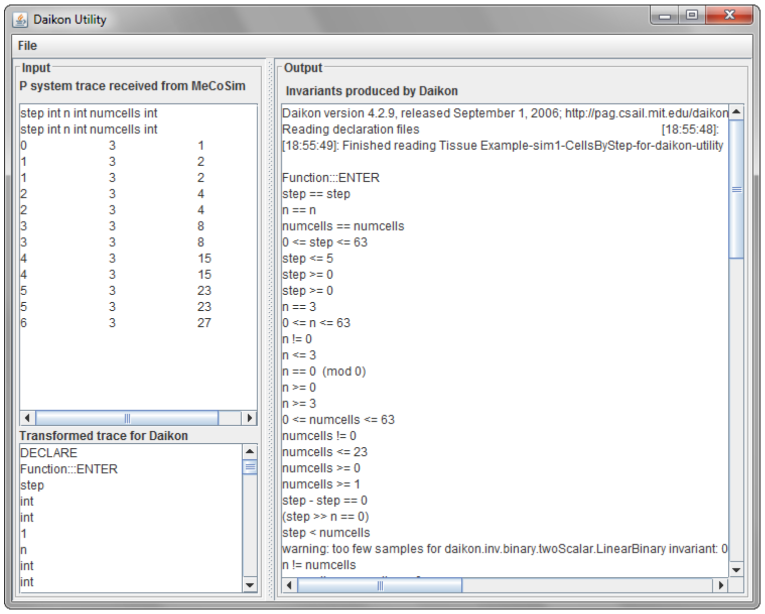
Once one of the extraction files is selected, the Daikon plugin runs from this input file. It automatically reads the simulation extraction file, generates the traces in the appropriate format and launches Daikon from this traces file, trying to detect as many invariants as possible. They are eventually visualized, as showed in figure 5.

Further technical details concerning the Daikon plugin and its integration into MeCoSim is provided in Appendix D.

4.4 Methodology

In the previous sections, different tools and languages for modelling, simulation, verification and invariant detection have been explained. In order to integrate the different tools in a systematic way, a novel methodology has been devised, as outlined in Fig. 1.

In the first stage, we need to model the problem, writing the model in P–Lingua, Promela and other languages, in order to serve as the default input for Spin and other possible model checkers.



P system trace received from MeCoSim

```
step int n int numcells int
step int n int numcells int
0        3        1
1        3        2
1        3        2
2        3        4
2        3        4
3        3        8
3        3        8
4        3        15
4        3        15
5        3        23
5        3        23
6        3        27
```

Transformed trace for Daikon

```
DECLARE
Function:::ENTER
step
int
int
1
n
int
int
```

Invariants produced by Daikon

```
Daikon version 4.2.9, released September 1, 2006; http://pag.csail.mit.edu/daikon
Reading declaration files                                            [18:55:48]:
[18:55:49]: Finished reading Tissue Example-sim1-CellsByStep-for-daikon-utility

Function:::ENTER
step == step
n == n
numcells == numcells
0 <= step <= 63
step <= 5
step >= 0
step >= 0
n == 3
0 <= n <= 63
n != 0
n <= 3
n == 0  (mod 0)
n >= 0
n >= 3
0 <= numcells <= 63
numcells != 0
numcells <= 23
numcells >= 0
numcells >= 1
step - step == 0
(step >> n == 0)
step < numcells
warning: too few samples for daikon.inv.binary.twoScalar.LinearBinary invariant: 0
n != numcells
```

Fig. 5. Daikon plugin - Invariants detection

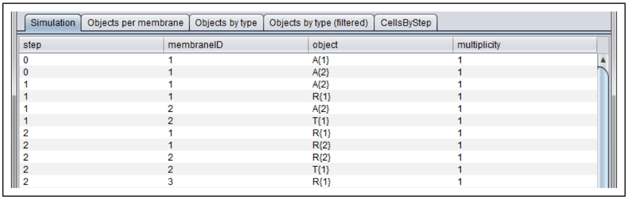
Once we have modelled the problem, written in P–Lingua, set the customized MeCoSim-based application including Daikon and entered the P–Lingua file in MeCoSim, the property extraction cycle can start.

For each iteration, some interesting goals should be addressed, for example to obtain relations between P system objects or invariants regarding the number of cells at each computation step. Then the needed outputs have to be set in MeCoSim, according to the stated objective. Based on the model and the target data entered in the input tables, simulations can be performed, obtaining the required results, which can be displayed and exported into the extraction files. The Daikon plugin can be executed further, in order to detect invariants from the extraction files, which contain traces from the P system simulations. Eventually, the new invariants detected by Daikon should be tested with the model checkers, to be verified.

In the following paragraphs this methodology is illustrated with some iterations for the 3–Col problem.

Iteration 1: property extraction using the entire model

The **goal** in this iteration is to analyze the values of all the objects across the simulation. A first output is set in MeCoSim (see Fig. 6), along with an extraction file for Daikon. No interesting properties (invariants) were found, so other alternative studies were considered.

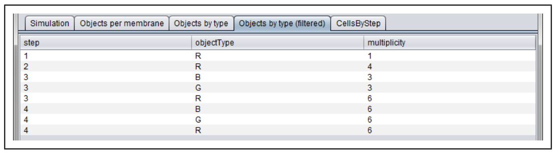


step	membraneID	object	multiplicity
0	1	A{1}	1
0	1	A{2}	1
1	1	A{2}	1
1	1	R{1}	1
1	2	A{2}	1
1	2	T{1}	1
2	1	R{1}	1
2	1	R{2}	1
2	2	R{2}	1
2	2	T{1}	1
2	3	R{1}	1

Fig. 6. Iteration 1. Output: entire simulation

Iteration 2: property extraction using the simplified model

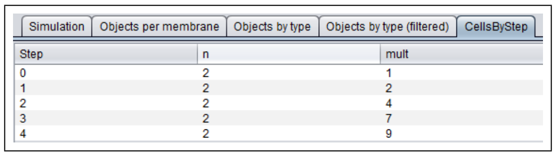
The **goal** in this iteration is to simplify the model and to filter the information sent to Daikon, for example by grouping the objects by type, or restricting the kind of objects analysed to R, G or B (and not considering the others, such as A and T). Again, the output is set in MeCoSim (see Fig. 7), along with an extraction for Daikon, and no interesting properties were found.



step	objectType	multiplicity
1	R	1
2	R	4
3	B	3
3	G	3
3	R	6
4	B	6
4	G	6
4	R	6

Fig. 7. Iteration 2. Output: objects by type (R, G, B)

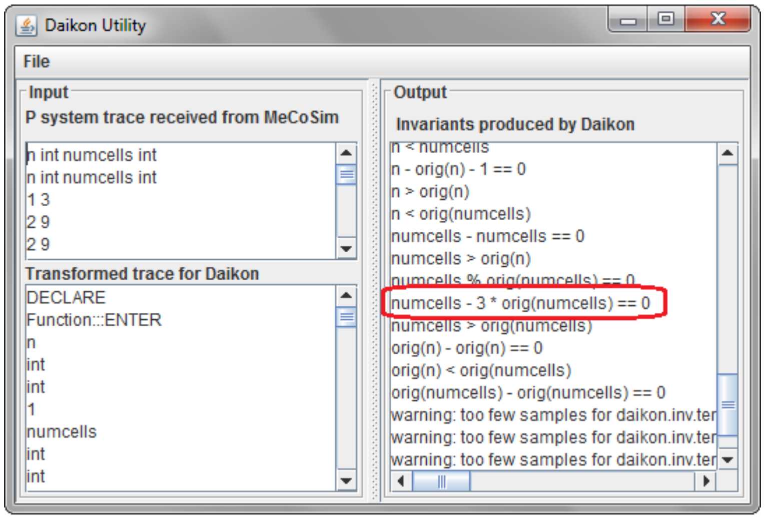
Another **goal** of this iteration is to obtain information (general formulas) regarding total number of cells in the P system for each computation step. To accomplish this, the respective output (see Fig. 8) and extraction are set. The simulation runs for $n = 2$, generating the extraction file, and Daikon Plugin is executed, but no relevant properties are obtained. The process is repeated for input $n = 3, 4, 5, \ldots$, but again no interesting properties are found.



Step	n	mult
0	2	1
1	2	2
2	2	4
3	2	7
4	2	9

Fig. 8. Iteration 2. Output: Cells by step

Instead of continuing with simulations for other values of n, it emerges the **idea** of collecting the results for different values of n in the same file, only for the last computation step, in order to get a general property, met for all the cases. The result is showed in figure 9. As it can be seen, an interesting invariant is detected: in the last configuration, having n as the input, $numcells = 3^n$ (Daikon indicates the equivalent recursive relation, $numcells - 3 * orig(numcells) == 0$).


Daikon Utility
File
Input
P system trace received from MeCoSim
Transformed trace for Daikon
Output
Invariants produced by Daikon

```
n int numcells int
n int numcells int
1 3
2 9
2 9
```

```
DECLARE
Function:::ENTER
n
int
int
1
numcells
int
int
```

```
n < numcells
n - orig(n) - 1 == 0
n > orig(n)
n < orig(numcells)
numcells - numcells == 0
numcells > orig(n)
numcells % orig(numcells) == 0
numcells - 3 * orig(numcells) == 0
numcells > orig(numcells)
orig(n) - orig(n) == 0
orig(n) < orig(numcells)
orig(numcells) - orig(numcells) == 0
warning: too few samples for daikon.inv.ter
warning: too few samples for daikon.inv.ter
warning: too few samples for daikon.inv.ter
```

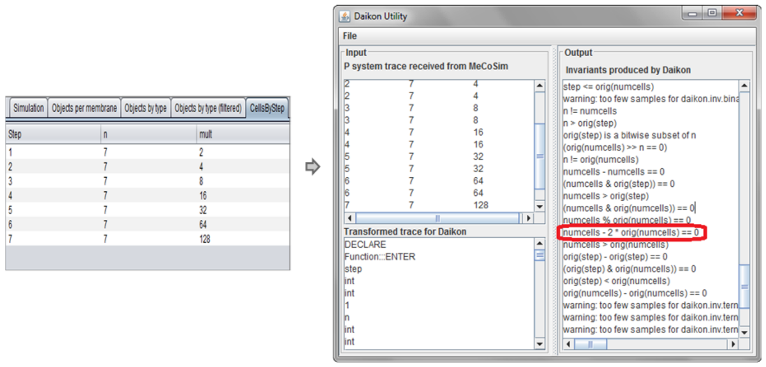
Fig. 9. Iteration 2. Invariants detection: Cells in last configuration

Iteration 3: dividing the computation path and extracting properties from the sub-paths, according to the model features

Another **idea** for extracting relevant information regarding the P system is to analyse only the configurations from a certain computation sub-path. In the case of the 3-Col problem, we anticipate that for the first half of the computation, the number of cells at each step is a power of 2.

The first **goal** in this iteration is to count the number of cells for each step until half the computation. The simulation runs, for example, for $n = 7$, generating the extraction file, and Daikon Plugin is executed, getting the results showed in Fig. 10. A new important invariant has been detected: for each $step$ until half the computation (that is, until $step = n$), $numcells = 2^{step}$ (Daikon indicates the equivalent recursive relation, $numcells - 2 * orig(numcells) == 0$).

The second **goal** in this iteration is to count the number of cells for each step in the second half of the computation. The output and extraction are set, the simulation runs again for $n = 7$, generating the extraction file. Daikon Plugin is executed, detecting the invariant: for each $step$ from half the computation to the end , $numcells = 3 (mod\, 4)$. However, this invariant was verified with the model checkers, resulting that this is not true for all the values of n, so it was not validated and cannot be considered a general property.


Simulation
Objects per membrane
Objects by type
Objects by type (filtered)
CellsByStep
Daikon Utility
File
Input
P system trace received from MeCoSim
Transformed trace for Daikon
Output
Invariants produced by Daikon

Step	n	mult
1	7	2
2	7	4
3	7	8
4	7	16
5	7	32
6	7	64
7	7	128

```
2        7        4
2        7        4
3        7        8
3        7        8
4        7        16
4        7        16
5        7        32
5        7        32
6        7        64
6        7        64
7        7        128
```

```
DECLARE
Function:::ENTER
step
int
int
1
n
int
int
```

```
step <= orig(numcells)
warning: too few samples for daikon.inv.bina
n != numcells
n > orig(step)
orig(step) is a bitwise subset of n
(orig(numcells) >> n == 0)
n != orig(numcells)
numcells - numcells == 0
(numcells & orig(step)) == 0
numcells > orig(step)
(numcells & orig(numcells)) == 0
numcells % orig(numcells) == 0
numcells - 2 * orig(numcells) == 0
numcells > orig(numcells)
orig(step) - orig(step) == 0
(orig(step) & orig(numcells)) == 0
orig(step) < orig(numcells)
orig(numcells) - orig(numcells) == 0
warning: too few samples for daikon.inv.tern
warning: too few samples for daikon.inv.tern
warning: too few samples for daikon.inv.tern
```

Fig. 10. Iteration 3. Invariants detection: Cells each last step, first half of the computation

As it has been seen, the methodology includes some important parts that have been integrated, automating the process of modelling, simulating, analyzing, debugging and detecting invariants from the MeCoSim application, making use of the simulation engine of pLinguaCore and the invariants detector Daikon. However, some parts of the process are being used independently, and have not been integrated with MeCoSim until the moment. The methodology covers all the process, and we plan to add the needed plugins to let it be as automated as possible.

4.5 Results - Summary of discovered properties

The idea of using simulation traces to infer properties of the P system model, as detected by Daikon, is useful in order to check the correct behaviour of the system and in the same time to find out new relationships between model variables. We can classify the results proposed by Daikon into:

- *Obvious invariants:* these confirm that the model is behaving as it should. For example, some results obtained for $n = 10$ are:
  - $B >= 0$, $B <= 196830$: the number of objects will never be negative, even more the sum of B_i objects over all membranes is at maximum 196830. The last relation is correct, moreover we estimate that the total number of occurrences of objects of type R_i, G_i, B_i at the end of computation is $n \times 3^{n-1}$ and $196830 = 10 \cdot 3^9$. This property has been verified with Spin for $n \in \{2, 3\}$ and with ProB for $n \in \{2, 3, 4\}$. For higher values of n, as stated in section 3.2 this property could not be verified because it involves checking many steps of the computations which yields to the "out of memory" problem.
  - $(step == 0) ==> (B == 0)$ is obvious because the initial multiset is $w_2 = A_1 \dots A_n$.

  - $B <= R$ is a direct consequence of the fact that rules $[A_i]_2 \rightarrow [R_i]_2[T_i]_2$ are applied first, so R_i objects are produced, and later are applied rules of type $[T_i]_2 \rightarrow [B_i]_2[G_i]_2$
  - $B >= orig(B)$ is obvious because B_i objects are never consumed by any rules.
- *Anomalous invariants:* these indicate a fault in the model and its parameter values. In this case, we did not obtained any of these, but if the P–Lingua model would have been incorrect (not solving the proposed problem), we could have encountered anomalous values.
- *Interesting invariants:* could even suggest novel relationships between the model variables. However, these properties should be further confirmed by a model checker verification.

A summary of the extracted properties is presented in Table 1, where by non-interesting properties we refer to obvious invariants, such the ones presented previously. Also, the truth value given in the third column refers to the result returned by the model checkers, after verifying the corresponding properties.

Extraction	**Result**	**Truth**
Entire model (all simulation data)	Non-interesting properties	
Objects grouped by type	Non-interesting properties	
Objects grouped by type, filtered by R, G or B	Non-interesting properties	
No. of cells for each n separately	Non-interesting properties	
No. of cells in the last configuration, for different values of n together	$numcells = 3^n$	true
No. of cells for each step $0 \ldots n$	$numcells = 2^{step}$	true
No. of cells for each step $(n+1) \ldots 2n$	$numcells = 3 (mod\, 4)$	false
No. of cells for each step $(n+1) \ldots (n + (n/2) + 1)$	$numcells = 3 (mod\, 12)$	false

Table 1. Summary of detected invariants

5 Discovering other properties with Maple

5.1 Model simplification for simulation

As we have discussed in Section 3.2, verification through model checking is possible for n up to 5. Beyond this point, due to the state explosion problem, both model checkers (Spin and ProB) will crash with an out of memory error, at least for some of the verified properties. This led us to build and use a simplified model of the investigated P system, based on the observation that the evolution of the number of cells and the actual contents of each cell are two separable issues, because only the symbols A and T appear in the left hand side of the rules. Thus, the number of objects R, G, B was ignored; first, a simplified Event-B model was produced, but with only limited performance gains, i.e. verification of properties for n up to 6; secondly, a Python implementation was developed, which enabled the verification of properties for n up to 19. Based on the backtracking technique, the simulation Algorithm 1 follows the evolution of a cell placed in the top of a stack, across the configurations of $\Pi(n)$ for as long as it contains non-terminals. Each cell is stored as a tuple containing the multiplicity of A's and T's and the step number. If the current cell still contains terminals, a new cell is added on to the stack, otherwise it is removed. We also used an array to count the number of cells produced at each step. The algorithm ends when the stack is empty. The verification Algorithm 2 checks if the values from the array calculated in the first algorithm are the same with the values returned by a function implemented for each property. Both algorithms are given in Appendix E.

5.2 Obtaining the polynomial coefficients with Maple

As we have seen in Section 4.4, using Daikon we have managed to find a number of (simpler) invariants but have failed to find other potentially interesting properties, such as the number of cells from configurations $n + 2$ up to configuration $2 * n$. This is due to the quite complex nature of the formulae for these numbers: Using the number of cells with label 2 given in Appendix B, we deduced that the number of cells in configuration $n + k$, $k \in \{2, 3, 4, 5, \ldots\}$, is a sum between 2^{n+k} and a polynomial $Q_k$of degree $k - 1$. In order to determine the exact expression of this polynomial we used Maple.

Maple is a powerful software that can be used to solve various mathematical problems with numerical and symbolic calculus. It also incorporates a programming language that allows working with formulas containing symbols and formal operations. Maple provides users over 5000 predefined functions and commands, with suggestive names, dedicated to various branches of mathematics.

In order to obtained Q_k we used the idea that a polynomial of degree $k - 1$ can be obtained solving a recurrence of order k and some Maple functions and commands:

- $rgf\_findrecur(k, seq, f, n)$: function contained in the package *genfunc* that finds the linear recurrence with constant coefficients of order k that is satisfied by the sequence with $2k$ terms *seq*; f is the name of the general term and n is the index variable of the recurrence;
- $rsolve(\{rec\}, f(n))$: command returning an expression for the general term of the function $f(n)$ by solving the recurrence *rec*;

- *expand*(*expr*): command used to distribute products over sums in the given expression.

For $k = 4$, using $rgf\_findrecur(4, [(81 - 2^8), (227 - 2^9), (585 - 2^{10}), (1403 - 2^{11}), (3185 - 2^{12}), (6947 - 2^{13}), (14729 - 2^{14}), (30619 - 2^{15})], f, t)$ we obtained the recurrence: $f(t) = 4 * f(t-1) - 6 * f(t-2) + 4 * f(t-3) - f(t-4)$. Solving this recurrence with the command $rsolve(\{f(t) = 4*f(t-1)-6*f(t-2)+4*f(t-3)-f(t-4), f(4) = -175, f(5) = -285, f(6) = -439, f(7) = -645\}, f(n))$ we obtain the general term: $-19+12(n+1)\left(\frac{1}{2}n+1\right)-14n-8(n+1)\left(\frac{1}{2}n+1\right)\left(\frac{1}{3}n+1\right)$, and expanding it: $-15 - \frac{32}{3}n - 2n^2 - \frac{4}{3}n^3$. Adding 2^{n+4} to this general term we obtain the number of cells with label 2 in the configuration $n+4$. The recurrences from Table 2 have been obtained similarly.

Configuration	Recurrence
$n+2$	$f(t) = 2 \cdot f(t-1) - f(t-2)$
$n+3$	$f(t) = 3 \cdot f(t-1) - 3 \cdot f(t-2) + f(t-3)$
$n+4$	$f(t) = 4 \cdot f(t-1) - 6 \cdot f(t-2) + 4 \cdot f(t-3) - f(t-4)$
$n+5$	$f(t) = 5 \cdot f(t-1) - 10 \cdot f(t-2) + 10 \cdot f(t-3) - 5 \cdot f(t-4) + f(t-5)$
$n+6$	$f(t) = 6 \cdot f(t-1) - 15 \cdot f(t-2) + 20 \cdot f(t-3)$ $-15 \cdot f(t-4) + 6 \cdot f(t-5) - f(t-6)$

Table 2. Recurrence for the number of cells with label 2 in configuration $n+k$

We notice that all these recurrences have, for each configuration $n+k$, the following form: $f(t) = \sum_{i=1}^{k}(-1)^{i+1} \cdot C_k^i \cdot f(t-i)$. Unfortunately, the current limitations of our tools ($n < 20$) does not allow us to continue the calculus with the next steps and so we cannot establish unequivocally if the recurrences have the same form for larger values of k.

Solving these recurrences (using *rsolve*), expanding the expressions (with *expand*) and adding 2^{n+k}, we obtain the number of cells with label 2 in the configuration $n+k$, for $k \in \{2,3,4,5,6\}$, as presented in Table 3.

All these formulas were verified using Algorithm 2 from Appendix E for $n < 20$. As we can see from Table 3 the coefficient of the polynomial that follows 2^{n+k} are rational but we could not establish any further rule for them except the fact that, in each case, the free term is $-2^k + 1$.

Configuration	Number of cells
$n+2$	$2^{n+2} - 2n - 3$
$n+3$	$2^{n+3} - 2n^2 - 4n - 7$
$n+4$	$2^{n+4} - \frac{4}{3} \cdot n^3 - 2 \cdot n^2 - \frac{32}{3} \cdot n - 15$
$n+5$	$2^{n+5} - \frac{2}{3} \cdot n^4 - \frac{28}{3} \cdot n^2 - 20 \cdot n - 31$
$n+6$	$2^{n+6} - \frac{4}{15} \cdot n^5 + \frac{2}{3} \cdot n^4 - \frac{20}{3} \cdot n^3 - \frac{32}{3} \cdot n^2 - \frac{676}{15} \cdot n - 63$

Table 3. Number of cells with label 2 in configuration $n + k$

6 Conclusion

In this paper, we have outlined an integrated methodology for P system formal verification, comprising modelling using P–Lingua, simulation with MeCoSim, properly extraction using Daikon and model checking using tools such as Spin and ProB. A plugin which allows Daikon to be called and used within MeCoSim has been developed and a (semi)-automatic Promela implementation has been generated from the P–Lingua model. A number of steps involved in property extraction using Daikon have been identified and the whole process has been illustrated with an example, a tissue P system model of the 3-colouring problem; this is a complex problem since, by using active membranes (cell division), the number of cells grows exponentially. As some of the sought properties have proved to be quite complex and could not be directly extracted using Daikon, a tool for mathematical and symbolic calculus (Maple) has been used to supplement our methodology.

Further work involves the development of completely integrated environment for automatic modelling, simulation and verification of P systems as well as applying the proposed methodology to other, more complex, P systems.

Acknowledgment

The work of MG, FI and RL was supported by a grant of the Romanian National Authority for Scientific Research, CNCS–UEFISCDI, project number PN-II-ID-PCE-2011-3-0688. The authors MPJ and LVC acknowledge the support of the project TIN2009–13192 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds, and the support of the Project of Excellence with Investigador de Reconocida Valía of the Junta de Andalucía, grant P08–TIC–04200. AT was supported by project POSDRU - "Dezvoltarea şcolilor doctorale prin acordarea de burse tinerilor doctoranzi cu frecvenţă"- 88/1.5/S/52826.

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Andrei, O., Ciobanu, G., Lucanu, D.: Executable specifications of P systems. In: Mauri, G., Păun, G., Pérez-Jiménez, M., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 3365, pp. 126–145. Springer Berlin / Heidelberg (2005)
3. Ben-Ari, M.: Principles of the Spin Model Checker. Springer-Verlag, London (2008)
4. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: Eleftherakis, G., Kefalas, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 4860, pp. 138–159. Springer (2007)
5. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Natural Computing Series, Springer (2006)
6. Ţurcanu, A., Ipate, F.: Modelling, testing and verification of P systems with active membranes using Rodin and ProB modelling. In: Proceedings of the 12th International Conference on Membrane Computing. pp. 459–468. Paris-Est University Press (2011)
7. Dang, Z., Ibarra, O., Li, C., Xie, G.: On model-checking of P systems. In: Calude, C., Dinneen, M., Păun, G., Pérez-Jímenez, M., Rozenberg, G. (eds.) Unconventional Computation, Lecture Notes in Computer Science, vol. 3699, pp. 82–93. Springer Berlin / Heidelberg (2005)
8. 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)
9. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A linear-time tissue P system based solution for the 3-coloring problem. Electr. Notes Theor. Comput. Sci. 171(2), 81–93 (2007)
10. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A logarithmic bound for solving Subset Sum with P systems. In: Eleftherakis, G., Kefalas, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 4860, pp. 257–270. Springer (2007)
11. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A uniform family of tissue P systems with cell division solving 3-COL in a linear time. Theor. Comput. Sci. 404(1-2), 76–87 (2008)
12. Díaz-Pernil, D., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A P-Lingua programming environment for membrane computing. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing - 9th International Workshop, WMC 2008, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 5391, pp. 187–203. Springer (2009)
13. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Software Eng. 27(2), 99–123 (2001)
14. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: An overview of P-Lingua 2.0. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing - 10th International Workshop, WMC 2009, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 5957, pp. 264–288. Springer (2010)

15. Gheorghe, M., Ipate, F., Lefticaru, R., Dragomir, C.: An integrated approach to P systems formal verification. In: CMC11: Proceedings of the Eleventh International Conference on Membrane Computing. pp. 225–238. ProBusiness Verlag Berlin (2010)
16. Ipate, F., Ţurcanu, A.: Modelling, verification and testing of P systems using Rodin and ProB. In: Ninth Brainstorming Week on Membrane Computing (BWMC 2011). pp. 209–220 (2011)
17. 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)
18. Ipate, F., Lefticaru, R., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Tudose, C.: Formal verification of p systems with active membranes through model checking. In: Gheorghe, M., Paun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Int. Conf. on Membrane Computing. Lecture Notes in Computer Science, vol. 7184, pp. 215–225. Springer (2011)
19. Ipate, F., Lefticaru, R., Tudose, C.: Formal verification of P systems using Spin. International Journal of Foundations of Computer Science 22(1), 133–142 (2011)
20. Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Colomer, M.A., Riscos-Núñez, A.: Mecosim: A general purpose software tool for simulating biological phenomena by means of p systems. IEEE Fifth International Conference on Bioinpired Computing: Theories and Applications (BIC-TA 2010) I, 637–643 (2010), `http://www.cs.us.es/~marper/investigacion/mecosim.pdf`
21. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
22. Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Tissue P system with cell division. In: Păun, G., Riscos-Núñez, A., Romero-Jiménez, A., Sancho-Caparrini, F. (eds.) Second Brainstorming Week on Membrane Computing. vol. Report RGNC 01/2004, pp. 308–386 (2004)
23. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
24. Romero-Campero, F.J., Gheorghe, M., Bianco, L., Pescini, D., Pérez-Jiménez, M.J., Ceterchi, R.: Towards probabilistic model checking on P systems using PRISM. In: Hoogeboom, H.J., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing - 7th International Workshop, WMC 2006, Revised, Selected, and Invited Papers. Lecture Notes in Computer Science, vol. 4361, pp. 477–495. Springer (2006)

Appendices

A Computation examples for $\Pi(n)$, $n \in \{2, 3\}$

Crt. step	**No. of cells labelled 2**	**Current configuration (only cells labelled with 2)**
0	1	$\left([A_1A_2]_2 \right)$
1	2	$\left([R_1A_2]_2\ [T_1A_2]_2 \right)$
2	4	$\left([R_1R_2]_2\ [R_1T_2]_2\ [B_1A_2]_2\ [G_1A_2]_2 \right)$
3	7	$\left([R_1R_2]_2\ [R_1B_2]_2\ [R_1G_2]_2\ [B_1R_2]_2\ [B_1T_2]_2\ [G_1R_2]_2\ [G_1T_2]_2 \right)$
4	9	$\left([R_1R_2]_2\ [R_1B_2]_2\ [R_1G_2]_2\ [B_1R_2]_2\ [B_1B_2]_2\ [B_1G_2]_2\ [G_1R_2]_2\ [G_1B_2]_2\ [G_1G_2]_2 \right)$

Table 4. Computation example for $\Pi(2)$

Crt. step	**No. of cells labelled 2**	**Current configuration (only cells labelled with 2)**
0	1	$\left([A_1A_2A_3]_2 \right)$
1	2	$\left([R_1A_2A_3]_2\ [T_1A_2A_3]_2 \right)$
2	4	$\left([R_1R_2A_3]_2\ [R_1T_2A_3]_2\ [B_1A_2A_3]_2\ [G_1A_2A_3]_2 \right)$
3	8	$\left([R_1R_2R_3]_2\ [R_1R_2T_3]_2\ [R_1B_2A_3]_2\ [R_1G_2A_3]_2\ [B_1R_2A_3]_2\ [B_1T_2A_3]_2\ [G_1R_2A_3]_2\ [G_1T_2A_3]_2 \right)$
4	15	$\left([R_1R_2R_3]_2\ [R_1R_2B_3]_2\ [R_1R_2G_3]_2\ [R_1B_2R_3]_2\ [R_1B_2T_3]_2\ [R_1G_2R_3]_2\ [R_1G_2T_3]_2\ [B_1R_2R_3]_2\ [B_1R_2T_3]_2\ [B_1B_2A_3]_2\ [B_1G_2A_3]_2\ [G_1R_2R_3]_2\ [G_1R_2T_3]_2\ [G_1B_2A_3]_2\ [G_1G_2A_3]_2 \right)$
5	23	$\left([R_1R_2R_3]_2\ [R_1R_2B_3]_2\ [R_1R_2G_3]_2\ [R_1B_2R_3]_2\ [R_1B_2B_3]_2\ [R_1B_2G_3]_2\ [R_1G_2R_3]_2\ [R_1G_2B_3]_2\ [R_1G_2G_3]_2\ [B_1R_2R_3]_2\ [B_1R_2B_3]_2\ [B_1R_2G_3]_2\ [B_1B_2R_3]_2\ [B_1B_2T_3]_2\ [B_1G_2R_3]_2\ [B_1G_2T_3]_2\ [G_1R_2R_3]_2\ [G_1R_2B_3]_2\ [G_1R_2G_3]_2\ [G_1B_2R_3]_2\ [G_1B_2T_3]_2\ [G_1G_2R_3]_2\ [G_1G_2T_3]_2 \right)$
6	27	$\left([R_1R_2R_3]_2\ [R_1R_2B_3]_2\ [R_1R_2G_3]_2\ [R_1B_2R_3]_2\ [R_1B_2B_3]_2\ [R_1B_2G_3]_2\ [R_1G_2R_3]_2\ [R_1G_2B_3]_2\ [R_1G_2G_3]_2\ [B_1R_2R_3]_2\ [B_1R_2B_3]_2\ [B_1R_2G_3]_2\ [B_1B_2R_3]_2\ [B_1B_2B_3]_2\ [B_1B_2G_3]_2\ [B_1G_2R_3]_2\ [B_1G_2B_3]_2\ [B_1G_2G_3]_2\ [G_1R_2R_3]_2\ [G_1R_2B_3]_2\ [G_1R_2G_3]_2\ [G_1B_2R_3]_2\ [G_1B_2B_3]_2\ [G_1B_2G_3]_2\ [G_1G_2R_3]_2\ [G_1G_2B_3]_2\ [G_1G_2G_3]_2 \right)$

Table 5. Computation example for $\Pi(3)$

B Number of cells labelled with 2

n	Number of cells labelled 2 at each configuration (from 0 to 17)																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	1	2	4	7	9													
3	1	2	4	8	15	23	27											
4	1	2	4	8	16	31	53	73	81									
5	1	2	4	8	16	32	63	115	179	227	243							
6	1	2	4	8	16	32	64	127	241	409	585	697	729					
7	1	2	4	8	16	32	64	128	255	495	891	1403	1867	2123	2187			
8	1	2	4	8	16	32	64	128	256	511	1005	1881	3185	4673	5857	6433	6561	
9	1	2	4	8	16	32	64	128	256	512	1023	2027	3891	6947	11043	15203	18147	19427
10	1	2	4	8	16	32	64	128	256	512	1024	2047	4073	7945	14729	24937	37289	48553
11	1	2	4	8	16	32	64	128	256	512	1024	2048	4095	8167	16091	30619	54395	87163
12	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8191	16357	32425	62801	115633
13	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16383	32739	65139	127651
14	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32767	65505	130617
15	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65535	131039
16	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131071
17	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
18	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
19	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072

Table 6. Number of cells labelled 2 at each configuration. Steps 0 to 17.

n	Number of cells labelled 2 at each configuration (from 18 to 25)							
	18	19	20	21	22	23	24	25
2 - 8								
9	19683							
10	55721	58537	59049					
11	123131	152827	169979	176123	177147			
12	195953	297457	399089	475633	516081	529393	531441	
13	241235	427219	689363	994003	1273811	1467347	1561555	1590227
14	257929	496537	909689	1543801	2372729	3261817	4014969	4496249
15	261627	519163	1012395	1902763	3363179	5460331	8007275	10538603
16	262109	523705	1042417	2050721	3927553	7168705	12186689	18927937
17	262143	524251	1047923	2089827	4135555	8028995	15023811	26524099
18	262144	524287	1048537	2096425	4185673	8315209	16300105	31081801
19	262144	524288	1048575	2097111	4193499	8378523	16686555	32930523

Table 7. Number of cells labelled 2 at each configuration. Steps 18 to 25.

n	Number of cells labelled 2 at each configuration						
	26	27	28	29	30	31	32
2 - 12							
13	1594323						
14	4713337	4774777	4782969				
15	12526187	13705835	14201451	14332523	14348907		
16	26553153	33603393	38758209	41613121	42735425	43013953	43046721
17	43301315	64409027	86709699	105964995	119129539	125961667	128484803
18	56571721	96349513	151011657	215527753	279384393	331059529	364220745
19	63682011	118735323	209492955	343710683	517551067	710439899	889828315

Table 8. Number of cells labelled 2 at each configuration. Steps 26 to 32.

n	Number of cells labelled 2 at each configuration					
	33	34	35	36	37	38
2 - 16						
17	129074627	129140163				
18	380408137	386044233	387289417	387420489		
19	1026339803	1108849627	1146860507	1159377883	1161999323	1162261467

Table 9. Number of cells labelled 2 at each configuration. Steps 33 to 38.

C P–Lingua model file

The content of the file in P–Lingua format, containing the specification of the model, is shown below.

```
@model<tissue_psystems>
def main()
{
/* tissue P system skeleton */
call example_tissue(n);
}
def example_tissue(n)
{
call init_cells();
call init_multisets(n);
call init_rules(n);
}
def init_cells()
{
@mu = [[]'2]'0;
}

def init_rules(n)
{
```

```
/* r1 */ [A{i}]'2 --> [R{i}]'2 [T{i}]'2 : 1<=i<=n;
/* r2 */ [T{i}]'2 --> [B{i}]'2 [G{i}]'2 : 1<=i<=n;
}

def init_multisets(n)
{
@ms(2) += A{i} : 1<=i<=n;
}
```

D Daikon integration in MeCoSim - plugin and config files

This appendix presents some technical details about the developed plugin, and the simple process of integration with MeCoSim.

The main code of the program is contained in `DaikonInterface.jar` program, that receives an input file with a compatible format for Daikon invariants detector. However, an additional jar file has been developed, `DaikonPlugin.jar`, to permit selecting among the different extraction files generated by the simulator.

To implement the integration of the program, the only work we have to do is the addition of a few lines in the file `plugins-properties` of MeCoSim, as follows:

```
plugin-daikon = daikonPlugin.Main
pluginname-daikon = Daikon
pluginmethod-daikon = pluginHook
pluginparam-daikon-1 = userfiles/daikon-files.txt
pluginjar-daikon-1 = DaikonInterface.jar
pluginjar-daikon-2 = DaikonPlugin.jar
```

E Algorithms

Algorithm 1 Calculating the number of membranes

```
function TriColor(n)
    $numberCellsStep[0..2n]$
    for i=0 do 2n-1
        $numberCellsStep[i] \leftarrow 0$
    end for
    $a \leftarrow n$
    $t \leftarrow 0$
    $step \leftarrow 0$
    stack.Push($a, t, step$)
    $numberCellsStep[step] \leftarrow numberCellsStep[step] + 1$
    while not stack.IsEmpty() do
        $a, t, step \leftarrow$ stack.Pop()
        $step \leftarrow step + 1$
        if $t > 1$ then
            $t \leftarrow t - 1$
            stack.Push($a, t, step$)
            stack.Push($a, t, step$)
            $numberCellsStep[step] \leftarrow numberCellsStep[step] + 1$
        else if $t = 1$ then
            if $a > 0$ then
                $t \leftarrow t - 1$
                stack.Push($a, t, step$)
                stack.Push($a, t, step$)
                $numberCellsStep[step] \leftarrow numberCellsStep[step] + 1$
            else
                $numberCellsStep[step] \leftarrow numberCellsStep[step] + 1$
            end if
        else if $a > 0$ then
            stack.Push($a - 1, t, step$)
            stack.Push($a - 1, t + 1, step$)
            $numberCellsStep[step] \leftarrow numberCellsStep[step] + 1$
        end if
    end while
```

```
    for i=1 do 2n-1
        $numberCellsStep[i] \leftarrow numberCellsStep[i - 1] + numberCellsStep[i]$
    end for
    return $numberCellsStep$
end function
```

Algorithm 2 Testing the invariants

```
function Fn2(n)
    return $2**(n+2) - 2*n - 3$
end function
function Fn3(n)
    return $2**(n+3) - 2*n**2 - 4*n - 7$
end function
function Fn4(n)
    $m1 \leftarrow n**3*4$
    $r1 \leftarrow m1 \mod 3$
    $m1 \leftarrow m1/3$
    $m2 \leftarrow 2*n**2$
    $m3 \leftarrow 32*n$
    $r3 \leftarrow m3 \mod 3$
    $m3 \leftarrow m3/3$
    $r \leftarrow (r1+r3)/3$
    return $2**(n+4) - m1 - m2 - m3 - r - 15$
end function
function Fn5(n)
    $m1 \leftarrow n**4*2$
    $r1 \leftarrow m1 \mod 3$
    $m1 \leftarrow m1/3$
    $m2 \leftarrow n**2*28$
    $r2 \leftarrow m2 \mod 3$
    $m2 \leftarrow m2/3$
    $m3 \leftarrow 20*n$
    $r \leftarrow (r1+r2)/3$
    return $2**(n+5) - m1 - m2 - m3 - r - 31$
end function
```

```
function Fn6(n)
    $m1 \leftarrow n**5*4$
    $r1 \leftarrow m1 \mod 15$
    $m1 \leftarrow m1/15$
    $m2 \leftarrow n**4*2$
    $m2 \leftarrow m2/3$
    $m3 \leftarrow n**3*20$
    $r3 \leftarrow m3 \mod 3$
    $m3 \leftarrow m3/3$
    $m4 \leftarrow n**2*32$
    $r4 \leftarrow m4 \mod 3$
    $m4 \leftarrow m4/3$
    $m5 \leftarrow n*676$
    $r5 \leftarrow m5 \mod 15$
    $m5 \leftarrow m5/15$
    $r \leftarrow (r3+r4)/3$
    $rr \leftarrow (r1+r5)/3$
    return $2**(n+6) - m1 + m2 - m3 - m4 - m5 - r - rr - 63$
end function
function Test(n)
    $a \leftarrow TriColor(n)$
    $nr \leftarrow len(a) - 1$
    $results \leftarrow []$
    if $n+2 \leq nr$ then
        $results.append(('n+2', a[n+2] == Fn2(n), Fn2(n))$
    end if
    if $n+3 \leq nr$ then
        $results.append(('n+3', a[n+3] == Fn3(n), Fn3(n))$
    end if
    if $n+4 \leq nr$ then
        $results.append(('n+4', a[n+4] == Fn4(n), Fn4(n))$
    end if
    if $n+5 \leq nr$ then
        $results.append(('n+5', a[n+5] == Fn5(n), Fn5(n))$
    end if
    if $n+6 \leq nr$ then
        $results.append(('n+6', a[n+6] == Fn6(n), Fn6(n))$
    end if
    return $results$
end function
```