kPWorkbench: A Software Framework for Kernel P Systems

Marian Gheorghe[1], Florentin Ipate[2], Laurentiu Mierla[2], and Savas Konur[1]

[1] School of Electrical Engineering and Computer Science, University of Bradford
Bradford BD7 1DP, UK
{m.gheorghe, s.konur}@bradford.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, laurentiu.mierla@gmail.com

Summary. *P* systems are the computational models introduced in the context of membrane computing, a computational paradigm within the more general area of unconventional computing. *Kernel P (kP)* systems are defined to unify the specification of different variants of P systems, motivated by challenging theoretical aspects and the need to model different problems. In this paper, we present kPWorkbench, a software framework developed to support kP systems. kPWorkbench integrates several *simulation* and *verification* tools and methods, and provides a software suit for the modelling and analysis of membrane systems.

1 Introduction

Membrane computing is a computational paradigm, within the more general area of unconventional computing [24], inspired by the structure and behaviour of the eukaryotic cell. The formal models introduced in this context are called membrane systems or P systems. After their introduction [22], membrane systems have been widely investigated for computational properties and complexity aspects, but also as a model for various applications [23]. The introduction of different variants of P systems has been motivated by challenging theoretical aspects, but also by the need to model different problems. An account of the theoretical developments is presented in [23], a set of general applications can be found in [6], whereas specific applications in systems and synthetic biology are provided in [11] and some of the future challenges are presented in [14]. More recently, applications in optimisations and graphics [16] and synchronisation of distributed systems [9] have been developed.

Several variants of P systems have been introduced and studied to model and analyse different problems, e.g., systems and synthetic biology [11], synchronisation of distributed systems [9], optimisations and graphics [16]. While the introduction of new variants allowed modelling different sets of problems, the ad-hoc addition

of new features has caused an abundance of P system variants, with a lack of a coherent integrating view, and well-defined framework would allow us to analyse, verify and validate the system behaviour.

We introduced *kernel P systems (kP systems)* [15] as an attempt to target these issues and create more general membrane computing models, integrating the most used concepts from P systems. A revised version of the model and the specification language can be found in [12] and its usage to specify the 3-colouring problem and a comparison to another solution provided in a similar context [8], is described in [13]. The kP systems have been also used to specify and analyse, through formal verification, synthetic biology systems [21, 20].

We have previously studied the theoretical aspects [15] and the verification and simulation techniques developed for kP systems [10, 3, 2]. In this paper, we present kPWorkbench (available and can be downloaded from its website `http://www.kpworkbench.org`), a software framework developed to support the analysis of kP systems. kPWorkbench integrates several *simulation* and *verification* tools and methods. The framework also facilitates verification by incorporating a property language based on *natural language* statements, which makes the property specification a very easy task. These features make kPWorkbench the only available tool supporting the non-probabilistic analysis of membrane systems through simulation and verification. The usability and novelty of our approach have been illustrated by some case studies [21, 20] chosen from synthetic biology (a new and emerging branch of biology that aspires to the engineering of new biological systems).

The paper is organised as follows: in Section 2 are introduced the key concepts and definitions related to kP systems; the kPWorkbench is discussed in Section 3; in Section 4 are summarised some kP systems applications; Section 5 illustrates through some examples the use of the kPWorkbench platform and final conclusions are provided in Section 6.

2 Kernel P Systems

A kP system is made of compartments placed in a graph-like structure. A compartment C_i has a type $t_i = (R_i, \sigma_i)$, $t_i \in T$, where T represents the set of all types, describing the associated set of rules R_i and the execution strategy that the compartment may follow. Note that, unlike traditional P system models, in kP systems each compartment may have its own rule application strategy. The following definitions are largely from [15].

Definition 1. *A* kernel P (kP) system *of degree n is a tuple*

$$k\Pi = (A, \mu, C_1, \ldots, C_n, i_0),$$

where A is a finite set of elements called objects*; μ defines the* membrane structure*, which is a graph, (V, E), where V are vertices indicating components, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a* compartment *of the system consisting of*

a compartment type from T and an initial multiset*, w_i over A; i_0 is the* output compartment *where the result is obtained.*

Each rule r may have a **guard** g denoted as $r\ \{g\}$. The rule r is applicable to a multiset w when its left hand side is contained into w and g holds for w. The guards are constructed using multisets over A and relational and Boolean operators. For example, rule $r : ac \to c\ \{\geq a^3 \wedge \geq b^2 \vee \neg > c\}$ can be applied iff the current multiset, w, includes the left hand side of r, i.e., ac and the guard holds for w - it has at least 3 a's and 2 b's or no more than a c. A formal definition may be found in [15].

Definition 2. *A rule associated with a compartment type l_i can have one of the following types:*

(a) **rewriting and communication** *rule: $x \to y\ \{g\}$,*
where $x \in A^+$ and y has the form $y = (a_1, t_1) \dots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j indicates a compartment type from T – see Definition 1 – with instance compartments linked to the current compartment; t_j might indicate the type of the current compartment, i.e., t_{l_i} – in this case it is ignored; if a link does not exist (the two compartments are not in E) then the rule is not applied; if a target, t_j, refers to a compartment type that has more than one instance connected to l_i, then one of them will be non-deterministically chosen;

(b) **structure changing rules***; the following types are considered:*

(b1) **membrane division** *rule: $[x]_{t_{l_i}} \to [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}}\ \{g\}$,*
where $x \in A^+$ and y_j has the form $y_j = (a_{j,1}, t_{j,1}) \dots (a_{j,h_j}, t_{j,h_j})$ like in rewriting and communication rules; the compartment l_i will be replaced by p compartments; the j-th compartment, instantiated from the compartment type t_{i_j} contains the same objects as l_i, but x, which will be replaced by y_j; all the links of l_i are inherited by each of the newly created compartments;

(b2) **membrane dissolution** *rule: $[x]_{t_{l_i}} \to \lambda\ \{g\}$;*
the compartment l_i and its entire contents is destroyed together with its links. This contrasts with the classical dissolution semantics where the inner multiset is passed to the parent *membrane - in a tree-like membrane structure;*

(b3) **link creation** *rule: $[x]_{t_{l_i}}; []_{t_{l_j}} \to [y]_{t_{l_i}} - []_{t_{l_j}}\ \{g\}$;*
the current compartment is linked to a compartment of type t_{l_j} and x is transformed into y; if more than one instance of the compartment type t_{l_j} exists then one of them will be non-deterministically picked up; g is a guard that refers to the compartment instantiated from the compartment type t_{l_i};

(b4) **link destruction** *rule: $[x]_{t_{l_i}} - []_{t_{l_j}} \to [y]_{t_{l_i}}; []_{t_{l_j}}\ \{g\}$;*
is the opposite of link creation and means that the compartments are disconnected.

Each compartment can be regarded as an instance of a particular *compartment type* and is therefore subject to its associated rules. One of the main distinctive features of kP systems is the execution strategy which is now statutory to types

rather than unitary across the system. Thus, each membrane applies its type specific instruction set, as coordinated by the associated execution strategy.

An execution strategy can be defined as a sequence $\sigma = \sigma_1 \& \sigma_2 \& \ldots \& \sigma_n$, where σ_i denotes an atomic component of the form:

- ϵ, an analogue to the generic *skip* instruction; ϵ is generally used to denote an *empty* execution strategy;
- r, a rule from the set R_t (the set of rules associated with type t). If r is applicable, then it is executed, advancing towards the next rule in the succession; otherwise, the compartment terminates the execution thread for this particular computational step and thus, no further rule will be applied;
- $(r_1, \ldots, r_n)$, with $r_i \in R_t, 1 \leq i \leq n$ symbolizes a non-deterministic choice within a set of rules. One and only one applicable rule will be executed if such a rule exists, otherwise the atom is simply skipped. In other words the non-deterministic choice block is always applicable;
- $(r_1, \ldots, r_n)^*$, with $r_i \in R_t, 1 \leq i \leq n$ indicates the arbitrary execution of a set of rules in R_t. The group can execute zero or more times, arbitrarily but also depending on the applicability of the constituent rules;
- $(r_1, \ldots, r_n)^\top, r_i \in R_t, 1 \leq i \leq n$ represents the maximally parallel execution of a set of rules. If no rules are applicable, then execution proceeds to the subsequent atom in the chain.

The execution strategy itself is a notable asset in defining more complex behaviour at the compartment level. For instance, weak priorities can be easily expressed as sequences of maximally parallel execution blocks: $(r_1)^\top \& (r_2)^\top \& \ldots \& (r_3)^\top$ or non-deterministic choice groups if single execution is required. Together with composite guards, they provide an unprecedented modelling fluency and plasticity for membrane systems. Whether such macro-like concepts and structures are preferred over traditional modelling with simple but numerous compartments in complex arrangements is a debatable aspect.

The kP system models are described in a machine readable language, called *kP–Lingua* [10]. Below, we illustrate the kP systems concepts on an example, which is slightly adjusted from [10, 2].

Example 1. A type definition in kP–Lingua.

```
type C1 {
    choice {
        > 2b : 2b -> b, a(C2) .
         b ->  2b .
    }
  }
  type C2 {
    choice {
      a -> a, {b, 2c}(C1) .
    }
 }
m1 {2x, b} (C1) - m2 {x} (C2) .
```

Above, `C1, C2` denote two compartment types, which are instantiated as `m1, m2`, respectively. `m1` starts with the initial multiset `2x, b` and `m2` starts with `x`. The rules of `C1` are chosen non-deterministically, only one at a time – this is achieved by the use of the key word `choice`. The first rule is fired only when its guard becomes true; in other words, only when the current multiset has at least three `b`'s. This rule also sends an `a` to the instance of `C2` that is linked. In the type `C2`, there is only one rule to be fired, which happens only when there is an `a` in the compartment `C1`.

3 kPWorkbench

kPWorkbench is an integrated software suit developed to provide a tool support for kP systems. kPWorkbench employs a set of tools and methods, allowing one to model membrane systems and to analyse them through *simulation* and *verification*. In the following, we briefly discuss some features of the software framework.

3.1 Features

Modeling.

kPWorkbench accepts kP system models specified in an intuitive modelling language, kP–Lingua. kP systems accumulate the most important aspects of various P system variants, so kP–Lingua provides a generic language to model various membrane systems. kPWorkbench features a graphical model editor, permitting to create new model files and editing existing files.

The grammar of the kP–Lingua language is written in ANTLR (ANother Tool for Language Recognition) [1], automatically generating the necessary syntactic and semantic analysers. ANTLR also constructs the data structures that represent the corresponding *abstract syntax tree* (AST) together with a traversing functionality.

Simulation.

kPWorkbench offers two different approaches to simulate kP systems. In both approaches, a kP–Lingua model is provided as an input, and the execution traces of the model are returned as an output. These traces permit exploring the dynamics of the system and observing how the system evolves over time.

In the first approach, we have developed a custom simulation tool [3], which recreates the system dynamics as a set of simulation runs. The tool translates a kP–Lingua specification into an internal data structure, which permits representing compartments, containing multisets of objects and rules, and their connections with other compartments.

In the second approach, we have integrated the Flame simulator [7], a general purpose large scale agent based simulation environment. Flame is based on the X-machine formalism [17], a type of extended finite state machine whose transitions

Prop. Pattern	Lang. Construct	LTL formula	CTL formula
Next	next p	X p	EX p
Existence	eventually p	F p	EF p
Absence	never p	$\neg$(F p)	$\neg$(EF p)
Universality	always p	G p	AG p
Recurrence	infinitely-often p	G F p	AG EF p
Steady-State	steady-state p	F G p	AF AG p
Until	p until q	p U q	A (p U q)
Response	p followed-by q	G ($p \rightarrow$ F q)	AG ($p \rightarrow$ EF q)
Precedence	p preceded-by q	$\neg(\neg p$ U $(\neg p \wedge q))$	$\neg$(E $(\neg p$ U $(\neg p \wedge q)))$

Table 1: Some property patterns defined in kP–Queries and the LTL and CTL translations. Note that LTL implicitly quantifies *universally* over paths (i.e. "necessity"). To complement this semantics, in CTL we translate some formulas by assuming quantification over *some* paths (i.e. "possibility").

are labelled by processing functions that operate on a (possibly infinite) set called memory, that models the system data. FLAME has been successfully used in various applications, ranging from biology to macroeconomics.

In order to simulate kernel P system models using the FLAME framework, an automated model translation has been implemented for converting the kP–Lingua specification into communicating X-machines [17]. One of the main advantages of this approach is the high scalability degree and efficiency for simulating large scale models.

Verification.

Although there have been some efforts to apply formal verification, in particular model checking, methods and methodologies for various P systems (e.g., [19, 4]), utilising a comprehensive, integrated and automated verification approach is a very challenging task in the context of membrane computing. For example, it is very difficult to transform some complex features, e.g. membrane division, dissolution and link creation/destruction, into suitable abstractions in model checking tools.

We have successfully addressed these issues, and developed a verification environment [10, 2] for kPWORKBENCH, integrating some state of the art model checking tools, e.g. the SPIN [18] and NUSMV [5]. The translations from a kP–Lingua representation to the corresponding SPIN and NUSMV inputs (i.e. PROMELA and SMV, respectively) are automatically performed.

In order to facilitate the property specification task, kPWORKBENCH features a property language, *kP–Queries*, based on *natural language* statements. The language also provides a list of property patterns (templates), generated from most commonly used queries (see Table 1). The property language permits specifying the target logic (i.e. LTL and CTL) for different properties without placing a requirement on a specific model checker. In this way, we can use the same set of properties in various verification experiments.

3.2 System architecture

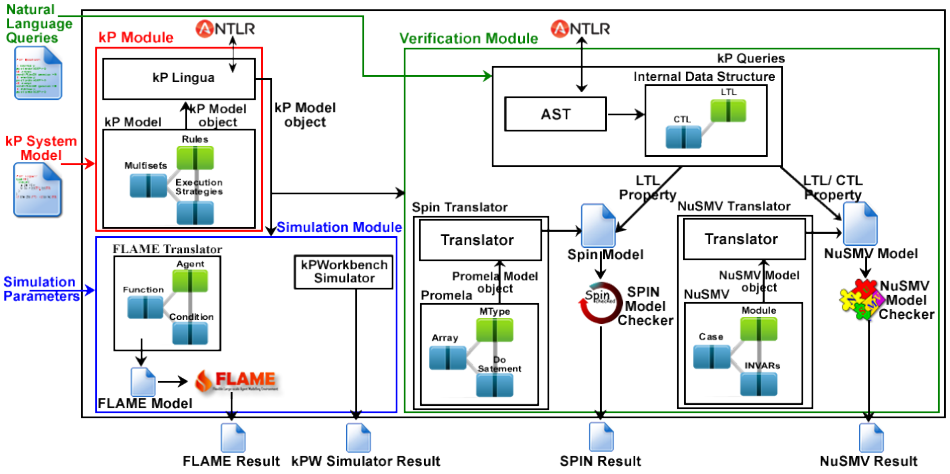

Fig. 1: The overview architecture of kPWorkbench framework

Figure 1 depicts an overview of the kPWorkbench system architecture, which consists of three modules:

1. The **kernel P (kP)** module takes a kP system model specified in kP–Lingua, which can be created or edited using a dedicated model editor, as input. The *kP–Lingua* module parses the input file and validates its syntax via ANTLR (which generates the necessary syntactic and semantic analysers). The *kP–Model* module accommodates the corresponding data structures of the input model, comprising compartment types, execution strategies, rules, multiset of objects and connections between compartments. The kP–Lingua module instantiates a kP–Model object and maps the AST generated by ANTLR to that object. This object is used as Data Transfer Object (DTO) between different modules of the framework. This separation helps developers to easily add new components to the framework.

2. The **Simulation** module consists of two components, kPWorkbench Simulator and Flame Translator. Both require the kP–Model object and simulator parameters, e.g. number of steps, as input. The kPWorkbench Simulator component is a custom simulator, which processes the multisets of objects of the input model with respect to its execution strategies and rules. The Flame Translator transforms the kP–Model object into a Flame Model object that aggregates agent, function, input, condition and output classes. It assigns each compartment to an agent, and the rules and the multiset of objects are stored as agent data. It creates a specific function for each type of execution strategy. In addition it creates C functions that represent the system behaviour (they are executed by Flame

when the agent makes a transition from one state to another). The FLAME Translator uses the ANTLR template group feature to produce the FLAME simulator specifications from the FLAME Model object.

3. The **Verification** module contains three components: the SPIN and NUSMV translators and the *kP–Queries* module:

The SPIN Translator has two main components: *Translator* and *Promela* (SPIN's specification language). The Promela component aggregates the Promela language specifications: *MType*, *Array*, *Do statement*, *If statement*, *Init*, etc. The Translator maps the kP–Model object to a *Promela* object using the following procedure [10]: (i) A compartment type is translated into a data type definition with the multiset of objects and links to other compartments, and also with temporary storage variables that provide the parallelism of P systems. (ii) Multiset of objects is assigned to an integer array where an index denotes the object and its value represents the multiplicity of the object. (iii) The set of rules are organised according to the execution strategies mapped by a *Proctype* definition – a Promela process. (iv) Maximal parallelism and arbitrary execution strategies are mapped to the *Do* statement, and choice execution strategy is mapped to *If* statement.

After the mapping process, the *Translator* component translates the Promela object to the corresponding Promela model, used by the SPIN model checker. However, this translation is not simple and straightforward, especially the structure changing rules, and arbitrary and maximal parallelism execution strategies complicate the translation process. More details about the translation from kP System model to the SPIN model checker specification can be found in [10].

Similarly, the NUSMV Translator translates the kP–Model object to the corresponding NUSMV representation (NUSMV's specification language). The translator has two main components: *Translator* and NUSMV. The NUSMV component consists of subcomponents representing the NUSMV language objects, such as *module*, *variables*, *INVARs*, *Case Statements*, *Conditions*, and *logical connectives*. The *Translator* maps the kP–Model object to the NUSMV object as follows: (i) Each compartment is translated into a module. (ii) The content of compartments is translated into variables. (iii) The initial multisets of the compartment are assigned into module parameters. (iv) Rules and guards are translated into the case statements. (v) The behaviour of execution strategies and the parallelism of P systems are achieved by introducing custom variables.

After the mapping process, the Translator component generates the NUSMV model from the NUSMV object, which is then provided as input to the NUSMV model checker. During the mapping process, we have overcome a few challenging domain specific restrictions. For example, unlike Promela, NUSMV has restrictions on defining arrays, and only allows accessing a value of array by a symbolic constant index; but it does not allow assigning a value by a symbolic constant. Therefore, instead of using arrays, we created a variable for each multiset of objects. Also, in Promela, we can non-deterministically pick a true statement among branches when there are more than one true statements; whereas, in NUSMV the selections are only deterministic. It always chooses the first true statement from a list of

conditions. We overcome that issue by introducing an *INVAR* declaration whenever a non-determinism behaviour is required.

The *kP–Queries* module receives a property, natural language based statements, as input. The user can build properties from the property language editor. The editor interacts with the kP–Lingua model, and permits accessing the native model elements, which simplifies the property building process. The kP–Queries' domain language has its own grammar, which is independent from and much simpler than the target model checking languages. The DSL (domain specific language) of the property language is written in ANTLR, receiving the EBNF grammar as input and generates the corresponding syntactic and semantic analysers as well as the corresponding AST. In order to simplify the traversal of the AST, we adapt a strategy, which maps the AST to a better structured internal data representation. To traverse between the elements of the internal data structure (a tree-like hierarchy), we follow the *Visitor design pattern*. Namely, the internal data nodes are treated as visitable entities, which are able to accept visitors and request to visit them. Each visitor has a specific functionality for visiting every single node. The visitor design pattern approach enables the kP–Queries module to translate every node of the internal presentation of property into the target model checker's corresponding property specification language.

4 Applications

Although membrane computing is mainly inspired from biology, its application to biological systems has been very limited due to the lack of a coherent and well-defined framework that allows us to analyse, verify and validate these systems. The methods and methodologies we have developed in [15, 10, 3, 2] to tackle these issues have filled an important gap in this respect. kPWorkbench, implementing these methodologies and algorithms, now provides a fully automated tool support, facilitating the modelling and analysis of biological systems through simulation and verification.

The usability and novelty of our approach has already been illustrated in some well-known case studies, chosen from systems and synthetic biology. In [21], we showed how our approach utilises the non-deterministic analysis of two biological systems, the quorum sensing in *P. aeruginosas* (a bacterial pathogen) and the synthetic pulse generator. Namely, we used our approach to observe various phenomena in genetic regulatory networks, e.g. various interactions between molecular species and various dependencies between molecules. Likewise, in [20], we showed how our approach can be used to formally analyse unconventional programs, e.g. some genetic Boolean gates.

We believe that our methods and techniques, and hence the kPWorkbench platform, provide significant contributions to the membrane & unconventional computing communities.

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin (LTL) Representations
1	Universality	**(i)** *No more than one termination signal will be generated*
		(ii) always m.t <= 1
		(iii) `ltl prop { [] (c[0].x[t_] <= 1 || state != step_complete) }`
2	Absence	**(i)** *The system will never generate 15 as a square number*
		(ii) never m.s = 15
		(iii) `ltl prop { !(<> (c[0].x[s_] == 15 && state == step_complete)) }`
3	Steady-state	**(i)** *In the long run, the system will converge to a state in which, if the termination signal is generated, no more a objects will be available*
		(ii) steady-state (m.a = 0 implies m.t = 1)
		(iii) `ltl prop { <> ([] ((c[0].x[a_] == 0 -> c[0].x[t_] == 1) ||`
		`state != step_complete) && state != step_complete) }`

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) NuSMV (CTL) Representations
4	Existence	**(i)** *The system will eventually consume all a objects, on some runs*
		(ii) eventually m.a = 0
		(iii) `SPEC EF m.a = 0`
5	Existence	**(i)** *On some runs the system will eventually halt*
		(ii) eventually m.t = 1
		(iii) `SPEC EF m.t = 1`
6	Universality	**(i)** *No more than one termination signal will be generated*
		(ii) always m.t <= 1
		(iii) `SPEC AG m.t <= 1`
7	Absence	**(i)** *The system will never generate 15 as a square number*
		(ii) never m.s = 15
		(iii) `SPEC !(EF m.s = 15)`
8	Precedence	**(i)** *The consumption of all a objects will always be preceded by a halting signal*
		(ii) m.a = 0 **preceded-by** m.t = 1
		(iii) `SPEC !(E [!(m.a = 0) U (!(m.a = 0) & m.t = 1)])`
9	Response	**(i)** *By starting the computation with at least one a object, on some runs the system will eventually consume all of them*
		(ii) m.a >0 followed-by m.a = 0
		(iii) `SPEC AG (m.a > 0 -> EF m.a = 0)`
10	Response	**(i)** *A halting signal will always be followed by the consumption of all a objects*
		(ii) m.t = 1 followed-by m.a = 0
		(iii) `SPEC AG (m.t = 1 -> EF m.a = 0)`

Table 2: List of properties derived from the property language and their representations in different formats.

5 Examples

5.1 Generating square numbers

We present below a kernel P systems model that generates square numbers (starting with 1) each step. The multiplicity of object "s" is equal to the square number produced each step.

```
type main {
max {
= t: a -> {} .
< t: a -> a, 2b, s .
< t: a -> a, s, t .
< t: b -> b, s .
```

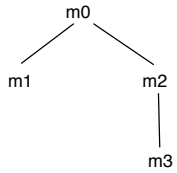

Fig. 2: The structure.

```
    }
}

m {a} (main) .
```

An execution trace for this model can be visualised as follows:

```
a
a 2b s
a 4b 4s
a 6b 9s
...
```

kPWorkbench automatically converts the kP-Lingua model into the corresponding input languages of the Spin, and NuSMV model checkers. In order to verify that the problem works as desired, we have constructed a set of properties specified in kP-Queries, listed in Table 2. The applied pattern types are given in the second column of the table. For each property we provide the following information; **(i)** informal description of each kP-Query, **(ii)** the formal kP-Query, **(iii)** the translated form of the kP-Query into the LTL specifications written in Spin modelling language, and CTL specifications written in the NuSMV language. The results of all queries are positive, as expected.

5.2 Broadcasting with acknowledgement

In this case study, we consider broadcasting with acknowledgement in ad-hoc networks. Each level of nodes in the hierarchy has associated a unique type with communication rules to neighbouring (lower and upper) levels. This is the only way we can simulate signalling with kP systems such that we do not hard-wire the target membranes in communication rules, i.e. assume we do not know how many child-nodes are connected to each parent as long as we group them by the same type; evidently, this only applies to tree structures. The kP Systems model written in kP–Lingua is given as follows:

```
type L0 {
max {
a -> b, a (L1), a (L2) .
    }
}
```

```
type L1 {
max {
a, c -> c (L0) .
    }
}

type L2 {
max {
a -> b, a (L3) .
b, c -> c (L0) .
    }
}

type L3 {
max {
a, c -> c (L2) .
    }
}

m0 {a} (L0) .

m1 {c} (L1) - m0 .
m2 {} (L2) - m0.
m3 {c} (L3) - m2 .
```

In order to verify that the model works as desired, we have verified some properties, presented in Table 3. The results are positive, except Properties 1 and 5, as expected. These results confirm the desired system behaviour.

6 Conclusion

We have presented the kPWorkbench toolset developed to support kernel P systems. kPWorkbench integrates several simulation and verification tools and methods and permits modelling and analysis of membrane systems. It also features a property language based on natural language statements to facilitate property specification. These features make kPWorkbench the only available integrated toolset permitting non-deterministic analysis (through simulation and verification) of membrane systems.

We are planning to work on more case studies from different fields, e.g., systems & synthetic biology, engineering and economics.

Acknowledgements. The work of FI and LM was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). MG and SK acknowledge the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1).

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin (LTL) Representations
3	Existence	**(i)** *The terminal nodes will receive the broadcast message at the same time*
		(ii) eventually (m1.a >0 and m3.a >0)
		(iii) `ltl prop { <> ((c[0].x[a_] > 0 && c[0].x[a_] > 0) && state == step_complete) }`
3	Absence	**(i)** *The root node will never receive an acknowledgement without sending a broadcast*
		(ii) never m0.a >0 and m0.c >0
		(iii) `ltl prop { !(<> ((c[0].x[a_] > 0 && c[0].x[c_] > 0) && state == step_complete)) }`
3	Response	**(i)** *The node m2 will always receive broadcast message before its child node (m3)*
		(ii) m2.a = 1 **followed-by** m3.a = 1
		(iii) `ltl prop { [] ((c[0].x[a_] == 1 -> <> (c[0].x[a_] == 1 && state == step_complete)) || state != step_complete) }`

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) NuSMV (CTL) Representations
4	Existence	**(i)** *The node m1 will eventually receive the broadcast message*
		(ii) eventually m1.a >0
		(iii) `SPEC EF m1.a > 0`
5	Existence	**(i)** *The terminal nodes will receive the broadcast message at the same time*
		(ii) eventually m1.a >0 and m3.a >0
		(iii) `SPEC EF (m1.a > 0 & m3.a > 0)`
6	Absence	**(i)** *The root node will never receive an acknowledgement without sending a broadcast*
		(ii) never m0.a >0 and m0.c >0
		(iii) `SPEC !(EF (m0.a > 0 & m0.c > 0))`
7	Response	**(i)** *The node m2 will always receive the broadcast message before its child node (m3)*
		(ii) m2.a = 1 **followed-by** m3.a = 1
		(iii) `SPEC AG (m2.a = 1 -> EF m3.a = 1)`
9	Steady-state	**(i)** *In the long run, the system will converge to a state in which the root node will have been received the acknowledgement from all the terminal nodes and no more broadcasts will occur*
		(ii) steady-state (m0.c = 2 implies m0.a = 0)
		(iii) `SPEC AF (AG (m0.c = 2 -> m0.a = 0))`
9	Steady-state	**(i)** *In the long run, the system will converge to a state in which the root node will have been received the acknowledgement from all the terminal nodes and no more acknowledgements will occur*
		(ii) steady-state (m0.c = 2 implies (m1.c = 0 and m3.c = 0))
		(iii) `SPEC AF (AG (m0.c = 2 -> (m1.c = 0 & m3.c = 0)))`

Table 3: List of properties derived from the property language and their representations in different formats.

References

1. ANTLR website, url: `http://www.antlr.org`
2. Bakir, M.E., Ipate, F., Konur, S., Mierlă, L., Niculescu, I.: Extended simulation and verification platform for kernel P systems. In: 15th International Conference on Membrane Computing. LNCS, vol. 8961, pp. 158–168. Springer (2014)
3. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. In: Proceedings of the 2014 IEEE 16th International Conference on High Performance Computing and Communication. pp. 409–412. HPCC '14, Paris, France (2014)
4. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F., Krasnogor, N., Gheorghe, M.: Infobiotics workbench: A P systems based tool for systems and synthetic biology. In: [11], pp. 1–41

5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An open source tool for symbolic model checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002). LNCS, vol. 2404, pp. 359–364. Springer, Copenhagen, Denmark (2002)
6. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Springer (2006)
7. Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C.: Exploitation of high performance computing in the FLAME agent-based simulation framework. In: Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication. pp. 538–545. HPCC '12, Liverpool, UK (2012)
8. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: A uniform family of tissue P systems with cell division solving 3-COL in a linear time. Theoretical Computer Science 404, 76–87 (2008)
9. Dinneen, M.J., Yun-Bum, K., Nicolescu, R.: Faster synchronization in P systems. Natural Computing 11(4), 637–651 (2012)
10. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierlă, L.: Model checking kernel P systems. In: 14th International Conference on Membrane Computing. LNCS, vol. 8340, pp. 151–172. Springer (2013)
11. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): Applications of Membrane Computing in Systems and Synthetic Biology. Springer (2014)
12. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - version 1. In: 11th Brainstorming Week on Membrane Computing, pp. 97–124. Fénix Editora (2013)
13. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M.J., Ţurcanu, A., Valencia-Cabrera, L., García-Quismondo, M., Mierlă, L.: 3-Col problem modelling using simple kernel P systems. Int. Journal of Computer Mathematics 90(4), 816–830 (2012)
14. Gheorghe, M., Păun, G., Pérez-Jiménez, M.J., Rozenberg, G.: Research frontiers of membrane computing: Open problems and research topics. International Journal of Foundations of Computer Scence 24, 547–624 (2013)
15. Gheorghe, M., Ipate, F., Dragomir, C.: Kernel P systems. In: 10th Brainstorming Week on Membrane Computing, pp. 153–170. Fénix Editora (2012)
16. Gimel'farb, G.L., Nicolescu, R., Ragavan, S.: P system implementation of dynamic programming stereo. Journal of Mathematical Imaging and Vision 47(1–2), 13–26 (2013)
17. Holcombe, M.: X-machines as a basis for dynamic system specification. Softw. Eng. J. 3(2), 69–76 (1988)
18. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Soft. Eng. 23(5), 275–295 (1997)
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. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. Fundamenta Informaticae 134(1-2), 97–110 (2014)
21. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology 4(1), 83–92 (2015)
22. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)

23. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
24. Rozenberg, G., Bäck, T., Kok, J.N. (eds.): Handbook of Natural Computing. Springer (2012)