Proceedings Paper:
Gheorghe, M., Konur, S., Ipate, F. et al. (3 more authors) (2015) An integrated model checking toolset for kernel P systems. In: Lecture Notes in Computer Science. 16th International Conference, CMC 2015, 17-21 Aug 2015, Valencia, Spain. Springer , pp. 153-170. ISBN 9783319284743

https://doi.org/10.1007/978-3-319-28475-0_11

An Integrated Model Checking Toolset for Kernel P Systems

Marian Gheorghe[1], Savas Konur[1], Florentin Ipate[2,3], Laurentiu Mierla[2,3], Mehmet E. Bakir[4], and Mike Stannett[4]

[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
[3] Department of Computer Science, University of Pitesti
Str. Targul din Vale, nr.1, 110040 Pitesti, Arges, Romania
[4] Department of Computer Science, University of Sheffield
Sheffield, S1 4DP, UK
mebakir1@sheffield.ac.uk

Abstract. *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. kP systems are supported by a software framework, called kPWorkbench, which integrates a set of related simulation and verification methodologies and tools. In this paper, we present an extension to kPWorkbench with a new model checking framework supporting the formal verification of kP system models. This framework supports both LTL and CTL properties. To make the property specification an easier task, we propose a property language, composed of natural language statements. We demonstrate our proposed methodology with an example.

1 Introduction

Membrane computing is a computational paradigm, within the more general area of unconventional computing [29], inspired by the structure and behaviour of eukaryotic cells. The formal models introduced in this context are called membrane systems or P systems. After their introduction [27], membrane systems have been widely investigated for computational properties and complexity aspects, but also as a model for various applications [28]. 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 [28], a set of general applications can be found in [6], whereas specific applications in systems and synthetic biology are provided in [11, 24, 20] and some of the future challenges are presented in [15]. More recently, applications in optimisations and graphics [16] and synchronisation of distributed systems [9] have been developed.

In many cases the specification of a certain system requires features, constraints or types of behaviour which are not always provided by a single formal model. It is very helpful to have some flexibility with modelling approaches. This flexibility might come from the way new features can be added or old ones are redefined. This approach might lead to a proliferation of various variants of the model. Software tools supporting the most used P system models have been conceived. They

come with a set of specification languages, known generically as P–Lingua [26]. P–Lingua aims to keep the syntax as close as possible to the original models and provides a simulation platform for all these models and a consistent user interface environment, called MeCoSim [25].

An alternative approach has been considered, by defining a specification language that allows to relatively easily specify the most utilised P system models. The newly defined concept of *kernel P systems* (*kP systems*) has been introduced in order to provide a theoretical support for this language. 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 [14]. The kP systems have been also used to specify and analyse, through formal verification, synthetic biology systems, e.g. genetic gates [22, 21].

Kernel P systems are supported by a software framework, kPWorkbench, which integrates a set of related simulation and verification methodologies and tools. In this paper, we present a new model checking framework that we have developed in support of formal verification of kernel P system models. The framework supports both LTL and CTL properties by making use of the Spin and NuSMV model checkers. To make the property specification an easier task, we propose a property language, composed of *natural language* statements. We demonstrate our proposed methodology on the subset sum problem.

The paper consists of five sections. Section 2 introduces the basic concepts related to kP systems. Section 3 discusses the previous model checking approach, and presents the new model checking methodology. Section 4 applies our proposed methodology to an instance of the subset sum problem. Section 5 briefly discusses the applicability of our approach to the analysis of biological systems. Finally, Section 6 draws conclusions and provides some future research directions.

2 Kernel P Systems

A kernel P system is a formal model that uses some well-known features of existing P systems and also includes some new concepts and, more importantly, it provides a coherent framework integrating all these elements. So, it can be considered as a unifying framework allowing to express different variants of P systems within the same formalism [12, 10, 2].

2.1 kP–Lingua

The kP system models are described in a machine readable language, called *kP–Lingua* [10]. Below, we illustrate the kP systems concepts with 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`.

2.2 kPWorkbench

The specifications written in kP–Lingua are supported by a software platform, kPWorkbench, which integrates a set of tools and translators that bridge several target specifications that we employ for kP system models, written in kP-Lingua. kPWorkbench permits *simulation* and *formal verification* of kP system models using several simulation and verification tools and methods.

The framework features a native simulator [3, 23], allowing the simulation of kP system models. In addition, it also integrates the Flame simulator [7], a general purpose large scale agent based simulation environment, based on a method that allows to express kP systems as a set of communicating X-machines [17].

kPWorkbench's model checking environment permits the formal verification of kernel P system models. The framework supports both *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)* properties by making use of the Spin [18] and NuSMV [5] model checkers. In order to facilitate the formal specification, kPWorkbench features a property language, called *kP-Queries*, comprising a list of natural language statements representing formal property patterns, from which the formal syntax of the Spin and NuSMV formulas are automatically generated.

3 Verification of kP systems

The application scope of P systems has recently broadened from contextual grammars to synthetic biology. This has unsurprisingly increased the efforts for establishing formal verification, in particular model checking, methods and methodologies for various P systems [19, 13, 4]. These successful attempts were mainly concerned with specific variants bound to an array of constraints, e.g. a limited feature set and a basic set of properties.

However, the efforts for a comprehensive, integrated and automated verification approach for general and unified languages, e.g. kP systems, are limited. This is mainly due to the computational challenges imposed by such formalisms. These bring in a lot of complications as they feature a dynamic structure by preserving the structure changing rules such as membrane division, dissolution and link creation/destruction. A state defined in this expansive context is consequently variable in size. It is, however, a challenging task to find the proper projections of such complex abstractions in model checking tools, as they require a fixed sized pre-allocated data model.

3.1 Previous Approach

In [10] we presented our initial efforts towards an integrated model checking approach, which permits formal properties to be verified against kP system models, specified in kP-Lingua, using the Spin model checker. The kP-Lingua representations of the models are automatically translated into the Spin's modelling language Promela. In order to ease the intricate and complex process of building logical formulas, the approach also features a *natural language query (NLQ)* tool, automatically converting predefined natural language queries into the corresponding Promela representation of temporal logic (LTL) formulas, through graphical user interface (GUI) elements.

Property Pattern	Language 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	q preceded-by p	$\neg(\neg p$ U $(\neg p \wedge q))$	$\neg$(E $(\neg p$ U $(\neg p \wedge q)))$

Table 1: The LTL and CTL property constructs currently supported by the kP-Queries file

In this approach, a strategy is devised to find a projection and mapping between a kP-Lingua model and the PROMELA representation. For some entities, e.g. multiset of objects, compartments, guards, rules, etc., finding a direct correspondence is possible. However, concepts such as maximal parallelism and membrane division are more difficult to deal with. To handle such cases, the following solution is devised [10]: "We collapse individual instructions (to atomic blocks) to the highest degree permitted by SPIN, minimizing the so-called intermediate state space which is irrelevant to a P system computation; and secondly, we appoint the states relevant to our model explicitly, using a global flag (i.e. a Boolean variable), raised when all processes have completed a computational step. Hence, we make a clear distinction between states that are pertinent to the formal investigation and the ones which should be discarded. This contrast is in turn reflected by the temporal logic formulae, which require adjustment to an orchestrated context where only a narrow subset of the global state space is pursued."

Although this approach employs useful strategies for both automatic translation of models and properties, it has some drawbacks: (i) It reformulates LTL properties into their corresponding PROMELA specifications. The translation requires introducing some special predicates into the state expressions. This results in long and complex state expressions, and hence formulas, which require manual manipulation of the corresponding translation in order to build complex queries with nested temporal operators. (ii) It only considers the use of SPIN model checker and hence only focuses on verifying LTL properties. Since there is no CTL model checker, e.g. NUSMV, integrated into the tool, we cannot verify CTL properties. (iii) According to some user feedbacks, the use of the NLQ tool has not been very practical. The tool has two user interfaces: one for constructing state expressions and one for constructing the actual properties. A property building task requires traversing between two interfaces, causing usability inconveniences.

3.2 The New Approach

To tackle these drawbacks, a new model checking environment for kPWORKBENCH has been developed, including a property language (with an editor) for the specification of queries[1] to be verified against kP-Lingua models. An EBNF grammar for this language is defined for the most common property patterns and a parser supporting the new property specification language has been implemented.

The property language editor interacts with the kP-Lingua model in question and allows users to directly access the native elements in the model, which results in less verbose and shorter state expressions, and hence more comprehensible formulas. These features and the natural language

[1] In the paper, we use the terms property and query interchangeably.

Pattern	Spin – LTL Translation	NuSMV – LTL Translation	NuSMV – CTL Translation
Next	`ltl p1 { X (!pInS U (`p` && pInS)) }`	`LTLSPEC X` p	`SPEC EX` p
Existence	`ltl p1 { <> (`p` && pInS) }`	`LTLSPEC F` p	`SPEC EF` p
Absence	`ltl p1 { !(<> (`p` && pInS)) }`	`LTLSPEC !(F` p`)`	`SPEC !(EF` p`)`
Universality	`ltl p1 { [] (`p` || pInS) }`	`LTLSPEC G` p	`SPEC AG` p
Recurrence	`ltl p1 { [] (<> (`p` && pInS) || !pInS) }`	`LTLSPEC G (F` p`)`	`SPEC AG (EF` p`)`
Steady-State	`ltl p1 { <> ([] (`p` || !pInS) && pInS) }`	`LTLSPEC F (G` p`)`	`SPEC AF (AG` p`)`
Until	`ltl p1 { (`p` || !pInS) U (`q` && pInS) }`	`LTLSPEC` p `U` q	`SPEC A [`p `U` q`]`
Response	`ltl p1 { [] ((`p` -> <> (`q` && pInS)) || !pInS) }`	`LTLSPEC G (`p `-> F` q`)`	`SPEC AG (`p `-> EF` q`)`
Precedence	`ltl p1 { !((!`p` || !pInS) U (!`p` && `q` && pInS)) }`	`LTLSPEC !(!`p `U (!`p `&` q`))`	`SPEC !(E [!`p `U (!`p `&` q`)])`

Table 2: The LTL and CTL property constructs currently supported by the kP-Queries file

like syntax of the language make the property construction much easier compared to our previous approach.

The new model checking environment supports both Spin and NuSMV model checkers. The translations from a kP-Lingua representation to the corresponding Spin and NuSMV inputs are automatically performed. The property language allows specifying the target logical formalism (i.e. LTL and CTL) for the different properties, without placing a requirement on a specific model checker, the same set of properties being able to be reused in various model checking experiments.

Targeting flexibility, expressivity and model checking language independence, the new verification approach for kP-Lingua models enriches kPWorkbench with a mechanism for defining *kP-Queries* files, which are especially designed for the purpose of being used to verify kP-Lingua models. The format of kP-Queries file is supported by an intuitive, coherent and integrated property specification language, allowing the construction of queries involving kP-Lingua model entities and targeting the LTL and CTL formalisms.

The new introduced property specification language aims to be independent from any target model checking language, yet integrating elements from LTL and CTL logical formalisms in a uniform way, such that property patterns from a set of most commonly used ones are considered in conjunction with two special keywords, *ltl* and *ctl*, giving the queries a formal context to be represented in. This approach also addresses one other limitation of the previous one, allowing the specification of nested properties in constructing more complex queries. Complex state expressions can be formulated by using relational and Boolean operators, while the only currently supported atomic operands are the object multiplicities of kP-Lingua model membranes. Table 1 summarizes the currently considered property patterns, together with the corresponding language construct, LTL and CTL representations.

Aiming for a generic and reusable property language, kP-Queries files do not embody any constructs that pertain to specific model checking languages, nor do they specify the target translation language the queries will be represented in. kP-Queries can be associated with kP-Lingua models, in conjunction with them serving as input for the translation engines defined in kPWorkbench. The properties specified in a formalism which is not supported by the target model checking language are simply discarded, only the appropriate ones being considered for translation.

kPWorkbench currently integrates translation mechanisms for two targets: Promela and Smv, the modeling languages of the model checkers Spin and NuSMV, respectively. While both Promela and Smv allow the specification of LTL properties, the latter also supports the CTL formalism.

As kP systems modelled in kP-Lingua are automatically translated into a computationally equivalent representation targeting a model checking language, the verification procedure should take into account one subtle difference concerning the modelling procedure and the underlying

formalism of the two computationally equivalent models. The translation of queries specified into the kP-Queries files needs to be formulated in such a way that we target only P system states (i.e. the states in which the computational step of the P system is completed), regardless of the various intermediate states required by the formalism of the translated model. This is the case for the translations targeting the SPIN model checker, as the translated model and properties are required to accommodate a special variable and state expressions over it, respectively. Namely, each LTL formula should be translated to SPIN using a special predicate, *pInS*, showing that the current SPIN state represents a P system configuration (the predicate is true when a SPIN configuration reaches a P system state on the execution path) or represents an intermediate state (it is false if intermediary steps are executed) – see [19] for the theoretical validation of this translation. On the other hand, the translations targeting NUSMV does not require a special treatment from the point of view of differentiating between source and destination model states. Table 2 depicts the translations of the above considered property patterns, targeting both SPIN and NUSMV, emphasizing also the use of the special Boolean variable *pInS*.

The implementation of the domain specific language used by kP-Queries files relies on ANTLR (ANother Tool for Language Recognition) [1] for its state of the art parser generator capabilities. The EBNF grammar of the property specification language serves as input for ANTLR in order to automatically generate the corresponding syntactical and semantic analyzers, together with the necessary data structures for representing the resulted *abstract syntax tree* (AST) and the underlying functionality of traversing it.

As the abstract syntax tree resulted from the parsing process directly reflects the structure of the grammar and its semantic model, a well defined domain model layer was introduced for supporting the internal representation of the data, thus decoupling the functionality relying on this data structure from the underlaying components of the parsing framework. By projecting the abstract syntax tree representation into a semantically equivalent internal data structure, a separation of concerns is achieved with the benefit of gaining greater flexibility in being able to independently change the parsing strategy from the property translation functionality. The projection of the abstract syntax tree to the internal data structure representation is achieved by the implementation of a model builder mechanism which is able to traverse the hierarchical representation of the AST, having at the same time the responsibility of semantically validating the kP-Queries files.

The new model checking module for kPWORKBENCH is especially designed around the concepts of maintainability and extensibility, following the SOLID programming principles [31] in achieving this goal. The entities composing the internal data representation, besides of playing the role of *data transfer objects* (DTO), are augmented with a minimal yet very powerful functionality for allowing them to be treated in an uniform way. The internal data structure is a tree-like hierarchical representation, augmented with the behavior required by the *Visitor design pattern* [32], aiming for *separation of concerns* (i.e. separating the translation strategies form the internal data structure they operate on) and following the *open/closed principle* (i.e. the set of translation strategies is open to be extended while the internal data structure is closed to further modifications).

The design pattern used in the model checking module implementation treats the nodes from the internal data representation as *visitable entities*, capable of accepting *visitors* and requests to visit them. Each *visitor* implementation holds specific functionality for visiting every single node. The model checking module implements its property translation strategies as visitors, being capable of translating every node of the internal representation of the properties into the corresponding form required by the target model checking language. By using this mechanism, each translation strategy implementation is independent, localized and coherent. Furthermore, a Singleton [30] implementation of a translation manager is able to receive an internal representation of a property

together with a translation target and to perform the translation of the property by instantiating the corresponding visitor and delegate it to visit the property data structure.

4 Case Studies

4.1 The Subset Sum problem

This case, the subset sum problem, will illustrate most of the features of the kP–Lingua, the presence of compartments, guarded rules and flexible execution strategies. The subset sum problem is stated as follows [2]:

Given a finite set $A = \{a_1, \ldots, a_n\}$, of n elements, where each element a_i has an associated weight, w_i, and a constant k, it is requested to determine whether or not there exists a subset $B \subseteq A$ such that $w(B) = k$, where $w(B) = \sum_{a_i \in B} w_i$. The following kP-Lingua code represents a model, where $n = 7$, $w(A) = \{3, 25, 8, 23, 5, 14, 30\}$ and $k = 55$.

```
type Main {

choice {
= 55x: a -> {yes, halt} (Output) .
> 55x: a -> # .
       }

choice {
!r1: a -> [a, r1][3x, a, r1] .
!r2: a -> [a, r2][25x, a, r2] .
!r3: a -> [a, r3][8x, a, r3] .
!r4: a -> [a, r4][23x, a, r4] .
!r5: a -> [a, r5][5x, a, r5] .
!r6: a -> [a, r6][14x, a, r6] .
!r7: a -> [a, r7][30x, a, r7] .
       }
}

type Output {
step -> 2step .
!yes: 9step -> no, halt .
}

main {a} (Main) - output {step} (Output) .
```

The model has two compartment types, `Main` and `Output`, and two compartments `output`, and `main`. The first rule of `Main` is a rewrite communication rule, which is guarded by {`= 55x`}. If this guard is satisfied, it will produce a `yes` and a `halt` object in `Output`, which is a positive answer for the problem. The second rule is a structure changing rule which results in the compartment dissolution. These two rules are encapsulated within a `choice` block, which means that at each step only one of the rules is selected and executed, and the selection is non-deterministic. The second `choice` block consists of seven division rules, each of which is guarded with $!\mathtt{r}_\mathtt{i}$, which aims to prevent any of the successor compartment to execute the same rule. Each rule divides the active compartment into two new compartments of the type `Main`. New compartments will inherit the

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin – LTL, (iv) NuSMV – LTL and (v) NuSMV – CTL Representations
1	Response	**(i)** *The execution of the computation will be followed by a halt*
		(ii) output.halt = 0 **followed-by** output.halt >0
		(iii) `ltl prop1 { [] ((m1[0].x[2] == 0 -><>(m1[0].x[2] >0 && state == step_complete) || state != step_complete) || state != step_complete) }`
		(iv) `LTLSPEC G (output.halt = 0 ->F output.halt >0)`
		(v) `SPEC AG (output.halt = 0 ->EF output.halt >0)`
2	Existence	**(i)** *The computation will eventually halt*
		(ii) **eventually** output.halt >0
		(iii) `ltl prop1 {<>(m1[0].x[2] >0 && state == step_complete)}`
		(iv) `LTLSPEC F output.halt >0`
		(v) `SPEC EF output.halt >0`
3	Until	**(i)** *The computation will eventually halt with either a 'yes' or 'no' result*
		(ii) output.halt = 0 **until** (output.halt >0 and (output.yes >0 or output.no >0))
		(iii) `ltl prop1 { (m1[0].x[2] == 0 || state != step_complete) U ((m1[0].x[2] >0 && (m1[0].x[3] >0 || m1[0].x[1] >0)) && state == step_complete) }`
		(iv) `LTLSPEC output.halt = 0 U (output.halt >0 & (output.yes >0 | output.no >0))`
		(v) `SPEC A [output.halt = 0 U (output.halt >0 & (output.yes >0 | output.no >0))]`
4	Until	**(i)** *The computation will halt within n+2 steps (for n=7)*
		(ii) (output.halt = 0 and output.step <= 9) **until** (output.halt >0 and output.step <=9)
		(iii) `ltl prop1 { ((m1[0].x[2] == 0 && m1[0].x[0] <= 9) || state != step_complete) U ((m1[0].x[2] > 0 && m1[0].x[0] <= 9) && state == step_complete) }`
		(iv) `(output.halt = 0 & output.step <= 9) U (output.halt > 0 & output.step <= 9)`
		(v) `A [(output.halt = 0 & output.step <= 9) U (output.halt > 0 & output.step <= 9)]`
5	Steady-state	**(i)** *The system will halt in the steady-state with a 'yes' or 'no' result*
		(ii) **steady-state** ((output.yes >0) or (output.no >0) implies (output.halt >0))
		(iii) `ltl prop1 { <>([] (((m1[0].x[3] >0 || m1[0].x[1] >0) ->m1[0].x[2] >0) || state != step_complete) && state == step_complete) }`
		(iv) `LTLSPEC F (G ((output.yes >0 | output.no >0) ->output.halt >0))`
		(v) `SPEC AF (AG ((output.yes >0 | output.no >0) ->output.halt >0))`
6	Absence	**(i)** *The computation will never halt with a 'no' result*
		(ii) **never** (output.halt >0 and (output.yes = 0 and output.no >0))
		(iii) `ltl prop1 { !(<>((m1[0].x[2] >0 && (m1[0].x[3] == 0 && m1[0].x[1] >0)) && state == step_complete)) }`
		(iv) `LTLSPEC !(F (output.halt >0 & (output.yes = 0 & output.no >0)))`
		(v) `SPEC !(EF (output.halt >0 & (output.yes = 0 & output.no >0)))`
7	Existence	**(i)** *A 'yes' result is eventually observed within no more than three steps*
		(ii) **eventually** (output.yes >0 and output.step <= 3)
		(iii) `ltl prop1 { <>((m1[0].x[3] >0 && m1[0].x[0] <= 3) && state == step_complete) }`
		(iv) `LTLSPEC F (output.yes >0 & output.step <= 3)`
		(v) `SPEC EF (output.yes >0 & output.step <= 3)`
8	Existence	**(i)** *A 'yes' result is eventually observed after more than three steps*
		(ii) **eventually** (output.yes >0 and output.step >3)
		(iii) `ltl prop1 { <>((m1[0].x[3] >0 && m1[0].x[0] >3) && state == step_complete) }`
		(iv) `LTLSPEC F (output.yes >0 & output.step >3)`
		(v) `SPEC EF (output.yes >0 & output.step >3)`
9	Precedence	**(i)** *A 'yes' result is always observed before a 'no' result*
		(ii) output.yes >0 **preceded-by** output.no >0
		(iii) `ltl prop1 { !((!(m1[0].x[3] >0) || state != step_complete) U (!(m1[0].x[3] >0) && m1[0].x[1] >0 && state == step_complete)) }`
		(iv) `LTLSPEC !(!(output.yes >0) U (!(output.yes >0) & output.no >0))`
		(v) `SPEC !(E [!(output.yes >0) U (!(output.yes >0) & output.no >0)])`

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

multiset objects of their parent. In addition, the multiset objects on the right hand side of the rule will pass to the corresponding child compartment. For example, if the first division rule is selected, then the compartment will be divided into two new compartments and both will inherit their parent objects. In addition, one of them will have the {`a, r1`} objects, while the other one will have {`3x, a, r1`}.

The compartment type `Output` has been added just to collect the results. We have extended the model to be able to produce a negative answer, `no`, if the system reaches its maximum number of steps and has not produced a positive answer, `yes`, so far. `Output` has two rewriting rules: the first rule increments the multiplicity of the `step` counter by one at each step, and the second rule produces a `halt` and a `no` object, if a `yes` object has not been produced so far, and the `step` counter is 9. In this case, both rules are executed (or at least the system attempts to execute both), in the given order.

kPWorkbench automatically converts the kP-Lingua model into the corresponding input languages of Spin, and NuSMV. In order to verify that the Subset Sum problem works as desired, we have constructed a set of properties specified in kP-Queries, listed in Table 3. A subset of these properties are verified in [10] using the model checker Spin using the old verification approach. Here, we use the new procedure of verifying kP-Lingua models for investigating the validity of a set of properties.

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 Spin modelling language, Promela, and into the **(iv)** CTL, and **(v)** LTL forms of the NuSMV specification. The results of all queries are positive.

In the following, we briefly describe why all properties listed in the Table 3 are true. After all division rules are applied, $2^7 = 128$ compartments of the type `Main` are generated. The contents of each compartment are determined by their ancestors. Here, we try to find out if any of the child compartments includes `55 x` objects. Since there are more than one compartment with exactly `55 x` objects, the output produces a `yes` and a `halt` object. If none of the compartments included `55 x` objects, then the output would produce a `no` and a `halt` object (without producing a `yes` object). Hence, a `halt` object is always produced. This explains why Properties 1, 2, 3, and 5 are true. Property 4 is also true, since the algorithm is a faithful linear time solution and the computation ends at most within $n + 2$ steps. Property 6 tests that there will never be a `no` object before a `yes` object is produced. Since there is at least one child compartment which has `55 x`, a `yes` will be triggered first. Thus, a `no` object is never produced before it. In other words, a `yes` object is always produced before a `no` object. Hence, Properties 6 and 9 are true. Property 7 is also true, because after the first step, the division rules will be applied and a child compartment will be created. Then, the child compartments that have `55 x` will trigger a `yes` object in the output. After the production of a `yes` object, it will remain inside the `output` compartment. This explains why Property 8 returns true.

As illustrated in Table 3, the intuitive and coherent form of kP-Queries lead to relatively short, yet natural language-like property specifications, which are independent from any specific model checking language. This approach brings the flexibility of independently considering the particularities of each model checker when translating properties, without the need to embody any of the required aspects into the specification language of kP-Queries. Being associated to kP-Lingua models, kP-Queries facilitates the construction of queries against the entities of the model and automatically considering the translation of these entities without user interaction.

Furthermore, unlike the previous one, the new verification approach is not bound to the usage of a graphical user interface. Although by using a GUI the property specification process is more intuitive, it is considered more tedious by most users aiming to script and automate a verification

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

task. In order to address this usability problem, the process of using the new kPWorkbench verification approach is also assisted by a simple GUI, guiding non-expert users through the entire procedure, while empowering experienced users with a flexible and expressive mechanism for using the verification framework from a command line interface or shell scripts.

4.2 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 .
```

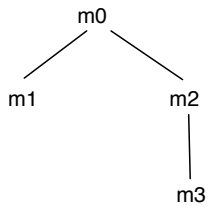

Fig. 1: The structure.

```
    }
}

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 4. 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.

4.3 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 5. The results are positive, except Properties 1 and 5, as expected. These results confirm the desired system behaviour.

5 Discussion

kP systems (and P systems in general) are a suitable formalism for modelling biological systems, especially multi-cellular systems and molecular interactions taking place in different locations of living cells. These non-deterministic models facilitate the *qualitative* analysis of such systems. Namely, they allow one to describe all chains of reactions, observe various interactions between species and determine various dependencies between molecules. In [22], two biological systems, the quorum sensing in *P. aeruginosas* and the synthetic pulse generator, and in [21] some genetic Boolean gates have been qualitatively analysed using the NuSMV and Spin model checkers. However, the ideas and methodology presented in these papers were not fully automated. Our work presented in this paper tackles this issue. The model checking framework now works in a fully automated fashion and is integrated into the kPWorkbench platform. Thus, this work can be considered progress on the conceptually presented methodology introduced in [22, 21].

6 Conclusions and Future Work

In this paper, we have presented a new model checking framework that we have developed in support of formal verification of kP system models. It supports both LTL and CTL properties by making use of the Spin and NuSMV model checkers. The new framework for kP-Lingua models enriches kPWorkbench with a mechanism for defining kP-Queries files, which are especially designed for the purpose of being used to verify kP-Lingua models. The format of kP-Queries

Prop.	Pattern	(i) Informal, (ii) Formal, (iii) Spin (LTL) Representations
1	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[1].x[a_] > 0 && c[3].x[a_] > 0) && state == step_complete) }`
2	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 the broadcast message before its child node (m3)*
		(ii) m2.a = 1 **followed-by** m3.a = 1
		(iii) `ltl prop { [] ((c[2].x[a_] == 1 -> <> (c[3].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)`
8	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 the all 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 5: List of properties derived from the property language and their representations in different formats.

file is supported by an intuitive, coherent and integrated property specification language, allowing the construction of queries involving kP-Lingua model entities and targeting the LTL and CTL formalisms. We have demonstrated our proposed methodology on the subset sum problem, by verifying a set of properties constructed in kP-Queries.

Recently, in addition to the properties presented in the paper, we have investigated and proved more complex and interested properties for the three examples provided. For example, in the square numbers generator, by introducing a new symbol to denote the iteration step and modifying the rules so that this is incremented whenever `s` is incremented from `a` (in rules (2) and (3) of type `Main`), we can verify that `s` equals the square of the iteration step. In the subset sum example, we cannot verify anything related to newly created compartments as we cannot refer to them. One way of addressing this problem would be to map somehow the compartment creation into a corresponding symbol in the Output compartment. For example, when a new compartment of type Main is created and this contains 3 elements (the first rule of the second choice of compartment Main), a rule which will send a `r1` into Output whenever a compartment with 3 elements is created (has both 3 `x` and r1) will be added. We can then verify that there is a path for which `output.r1 > 0` (this means in Output an `r1` has been received after the compartment `Main` with 3 elements has been created).

We aim to extend the current implementation by considering more complex queries over kP-Lingua model entities, offering the verification tool greater power and expressivity, and also by investigating how properties involving the active membranes can be formulated and proved. We also aim to evaluate the methodology with several other case studies to better understand its potential and limitations more generally. In this respect, we will expand the synthetic biology investigations [22, 21] and develop verification strategies for some synchronisation [9] and graphics [16] problems.

Acknowledgements.

SK and MG acknowledge the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1). 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). MB is supported by a PhD studentship provided by the Turkey Ministry of Education.

References

1. ANTLR website. url: `http://www.antlr.org`.
2. M. E. Bakir, F. Ipate, S. Konur, L. Mierlă, and I. Niculescu. Extended simulation and verification platform for kernel P systems. In *15th International Conference on Membrane Computing*, volume 8961 of *LNCS*, pages 158–168. Springer, 2014.
3. M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High performance simulations of kernel P systems. In *Proceedings of the 2014 IEEE 16th International Conference on High Performance Computing and Communication*, HPCC'14, pages 409–412, Paris, France, 2014.
4. J. Blakes, J. Twycross, S. Konur, F. Romero-Campero, N. Krasnogor, and M. Gheorghe. Infobiotics workbench: A P systems based tool for systems and synthetic biology. In *Applications of Membrane Computing in Systems and Synthetic Biology*, volume 7 of *Emergence, Complexity and Computation*, pages 1–41. Springer, 2014.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An open source tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 359–364, Copenhagen, Denmark, 2002. Springer.
6. G. Ciobanu, M. J. Pérez-Jiménez, and G. Păun, editors. *Applications of Membrane Computing*. Springer, 2006.
7. S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. 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*, HPCC'12, pages 538–545, Liverpool, UK, 2012.
8. D. Díaz-Pernil, M. A. Gutiérrez-Naranjo, and M. J. Pérez-Jiménez. 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. M. J. Dinneen, K. Yun-Bum, and R. Nicolescu. Faster synchronization in P systems. *Natural Computing*, 11(4):637–651, 2012.
10. C. Dragomir, F. Ipate, S. Konur, R. Lefticaru, and L. Mierlă. Model checking kernel P systems. In *14th International Conference on Membrane Computing*, volume 8340 of *LNCS*, pages 151–172. Springer, 2013.
11. P. Frisco, M. Gheorghe, and M. J. Pérez-Jiménez, editors. *Applications of Membrane Computing in Systems and Synthetic Biology*. Springer, 2014.
12. M. Gheorghe, F. Ipate, C. Dragomir, L. Mierlă, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez. Kernel P systems - version 1. In *11th Brainstorming Week on Membrane Computing*, pages 97–124. Fénix Editora, 2013.

13. M. Gheorghe, F. Ipate, R. Lefticaru, and C. Dragomir. An integrated approach to P systems formal verification. In *Membrane Computing*, volume 6501 of *LNCS*, pages 226–239. Springer, 2011.
14. M. Gheorghe, F. Ipate, R. Lefticaru, M. J. Pérez-Jiménez, A. Ţurcanu, L. Valencia-Cabrera, M. García-Quismondo, and L. Mierlă. 3-Col problem modelling using simple kernel P systems. *Int. Journal of Computer Mathematics*, 90(4):816–830, 2012.
15. M. Gheorghe, G. Păun, M. J. Pérez-Jiménez, and G. Rozenberg. Research frontiers of membrane computing: Open problems and research topics. *International Journal of Foundations of Computer Scence*, 24:547–624, 2013.
16. G. L. Gimel'farb, R. Nicolescu, and S. Ragavan. P system implementation of dynamic programming stereo. *Journal of Mathematical Imaging and Vision*, 47(1–2):13–26, 2013.
17. M. Holcombe. X-machines as a basis for dynamic system specification. *Softw. Eng. J.*, 3(2):69–76, 1988.
18. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Soft. Eng.*, 23(5):275–295, 1997.
19. F. Ipate, R. Lefticaru, and C. Tudose. Formal verification of P systems using Spin. *International Journal of Foundations of Computer Science*, 22(1):133–142, 2011.
20. S. Konur and M. Gheorghe. A property-driven methodology for formal analysis of synthetic biology systems. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, volume 12, pages 360–371, 2015.
21. S. Konur, M. Gheorghe, C. Dragomir, F. Ipate, and N. Krasnogor. Conventional verification for unconventional computing: a genetic XOR gate example. *Fundamenta Informaticae*, 134(1-2):97–110, 2014.
22. S. Konur, M. Gheorghe, C. Dragomir, L. Mierlă, F. Ipate, and N. Krasnogor. Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. *ACS Synthetic Biology*, 4(1):83–92, 2015.
23. S. Konur, M. Kiran, M. Gheorghe, M. Burkitt, and F. Ipate. Agent-based high-performance simulation of biological systems on the Gpu. In *Proceedings of the 2015 IEEE 15th International Conference on High Performance Computing and Communication*, HPCC'15, New York, USA, 2015.
24. S. Konur, C. Ladroue, H. Fellermann, D. Sanassy, L. Mierlă, F. Ipate, S. Kalvala, M. Gheorghe, and N. Krasnogor. Modeling and analysis of genetic boolean gates using the infobiotics workbench. In *Proceedings of Verification of Engineered Molecular Devices and Programs*, VEMDP'14, pages 26–37, Vienna, Austria, 2014.
25. MeCoSim website. url: `http://www.p-lingua.org/mecosim/`.
26. P-Lingua website. url: `http://www.p-lingua.org`.
27. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
28. G. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
29. G. Rozenberg, T. Bäck, and J. N. Kok, editors. *Handbook of Natural Computing*. Springer, 2012.
30. Singleton. url: `http://en.wikipedia.org/wiki/Singleton_pattern`.
31. SOLID. url: `http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)`.
32. Visitor design pattern. url: `http://en.wikipedia.org/wiki/Visitor_pattern`.