Model Checking Kernel P Systems

Ciprian Dragomir[1], Florentin Ipate[2,3], Savas Konur[1], Raluca Lefticaru[2,3], and Laurentiu Mierla[3]

[1] Department of Computer Science, University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
c.dragomir@sheffield.ac.uk, s.konur@sheffield.ac.uk

[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, raluca.lefticaru@fmi.unibuc.ro

[3] Department of Mathematics and Computer Science, University of Piteşti
Str. Târgu din Vale 1, 110040, Piteşti, Romania
laurentiu.mierla@gmail.com

Abstract. Recent research in membrane computing examines and confirms the anticipated modelling potential of kernel P systems in several case studies. On the one hand, this computational model is destined to be an abstract archetype which advocates the unity and integrity of P systems onto a single formalism. On the other hand, this envisaged convergence is conceived at the expense of a vast set of primitives and intricate semantics, an exigent context when considering the development of simulation and verification methodologies and tools. Encouraged and guided by the success and steady progress of similar undertakings, in this paper we directly address the issue of formal verification of kernel P systems by means of model checking and unveil a software framework, *kpWorkbench*, which integrates a set of related tools in support of our approach. A case study that centres around the well known *Subset Sum* problem progressively demonstrates each stage of the proposed methodology: expressing a kP system model in recently introduced *kP-Lingua*; the automatic translation of this model into a Promela (Spin) specification; the assisted, interactive construction of a set of LTL properties based on natural language patterns; and finally, the formal verification of these properties against the converted model, using the Spin model checker.

1 Introduction

Membrane computing, the research field introduced by Gheorghe Păun [21], studies computational models, called P systems, inspired by the functioning and structure of the living cell. In recent years, significant progress has been made in using various types or classes of P systems to model and simulate systems and problems from many different areas [5]. However, in many cases, the specifications developed required the ad-hoc addition of new features, not provided in the initial definition of the given P system class. While allowing more flexibility in modelling, this has led to a plethora of P system variants, with no coherent integrating view, and sometimes even confusion with regard to what variant or functioning strategy is actually used.

The concept of *kernel P system* (*kP system*) [7] has been introduced as a response to these problems. It integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, thus, provides a framework for formally analyzing these models. The expressive power and efficiency of the newly introduced kP systems have been illustrated by a number of representative case studies [8, 14]. Furthermore, the kP model is supported by a modelling language, called *kP-Lingua*, capable of mapping the kernel P system specification into a machine readable representation.

Naturally, formal modelling has to be accompanied by formal verification methods. In the membrane system context, formal verification has been approached, for example, using rewriting logic and the Maude tool [1] or PRISM and the associated probabilistic temporal logic [11] for stochastic systems [3]. Several, more recent, successful attempts to apply model checking techniques on transition P systems also exist [4, 17, 18, 15]. However, to the best of our knowledge, there is no integrated formal verification approach to allow formal properties to be specified in a language easily accessible to the non-specialist user and to be automatically verified in a transparent way.

This paper proposes precisely such an integrated verification approach, which allows formal properties, expressed in a quasi-natural language using predefined patterns, to be verified against a kP-Lingua representation of the model using model checking techniques and tools (in this case the model checker Spin and the associated modelling language Promela). Naturally, this approach is supported by adequate tools, which automatically convert the supplied inputs (natural language queries and kP-Lingua representation) into their model checking specific counterparts (LTL queries and Promela representation, respectively). The approach is illustrated with a case study, involving a kP system solving a well-known NP-complete problem, the Subset Sum problem.

The paper is structured as follows: Section 2 recalls the definition of a kernel P system - the formal modelling framework central to our examination. We then review, in section 3, some of the primary challenges of model checking applicable to kP system models and discuss the transformations such a model must undergo, in order to be exhaustively verified by Spin. We also present our implemented approach to achieve an automatic model conversion, targeting the process meta language, *Promela*. In section 4, we address the complementary requirement of specifying system properties as temporal logic formulae. The section also includes an array of EBNF formal definitions which describe the construction of LTL properties that relate to kP system state constituents, a guided process which employs selected natural language query patterns.

Section 5 applies our proposed methodology, exemplifies and demonstrates all stages of the process with a case study - an instance of the Subset Sum problem. Finally, we conclude our investigation and review our findings 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 [7].

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

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

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 is true for w. The guards are constructed using multisets over A and relational and Boolean operators. For example, rule $r : ac \rightarrow c \ \{\geq a^3 \wedge \geq b^2 \vee \neg > c\}$ can be applied iff the current multiset, w, includes the left hand side of r, i.e., ac and the guard is true 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 [7].

Definition 2. *A rule can have one of the following types:*

- *(a)* **rewriting and communication** *rule: $x \rightarrow y \ \{g\}$,*
  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: $[x]_{t_{l_i}} \rightarrow [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}} \ \{g\}$,*
    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: $[x]_{t_{l_i}} \rightarrow \lambda \ \{g\}$;*
    the compartment l_i and its entire contents is be 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:* $[x]_{t_{l_i}}; []_{t_{l_j}} \rightarrow [y]_{t_{l_i}} - []_{t_{l_j}}\ \{g\}$*;*
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:* $[x]_{t_{l_i}} - []_{t_{l_j}} \rightarrow [y]_{t_{l_i}}; []_{t_{l_j}}\ \{g\}$*;*
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 Kernel P 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; *epsilon* 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, execution halts, pruning the remainder of the sequence;
- $(r_1, \ldots, r_n)$, 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, 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.

3 kP system models and the Spin model checker

Formal verification of P systems has become an increasingly investigated subject, owing to a series of multilateral developments which have broaden its application scope and solidified some domain specific methodologies. Although there have been several attempts that successfully demonstrated model checking techniques on P systems ([17], [18], [15]), the analysis is always bound to an array of constraints, such as specific P system variants with a limited feature set and a very basic set of properties. Nevertheless, there are notable advancements which have paved the path towards a more comprehensive, integrated and automated approach we endeavour to present in this paper.

The task of P system model checking is perhaps a most inviting and compelling one due to the many onerous challenges it poses. On the one hand we are confronted with the inherent shortcomings of the method itself, which have a decisive impact on the tractability of some models and, in the best case, the efficiency or precision of the result is severely undermined. Speaking generally, but not inaccurately, model checking entails an exhaustive, *strategic* exploration of a model's state space to assert the validity of a logically defined property. Hence, the state space is of primary concern and we can immediately acknowledge 1. the requirement for models to have a finite state space and 2. the proportionality between the state space *size* and the stipulated computational resources, which ultimately determines the feasibility of the verification process.

On the other hand, the complex behaviour of certain computational models translates to elaborate formal specifications, with intricate semantics and more often than not, a vast set of states. However, it is the tireless state explosion problem that diminishes the applicability of model checking to concurrent systems, a rather ironical fact, since such systems are now the primary target for exhaustive verification.

We shall not delve any further into general aspects since our focus is not the vivisection of a methodology, but rather the introduction of a robust, integrated and automated approach that constellates around kernel P systems and overtly addresses the predominant challenges of model checking emphasised so far.

The three most conspicuous features that typify membrane systems are 1. a structured, distributed computational environment; 2. multisets of objects as atomic terms in rewriting rules and 3. an *execution strategy* according to which the rules are applied. We recall that kP systems explicitly associate an instruction set to an array of compartments employing the *type - instance* paradigm. As it turns out, this distinction is highly relevant in mapping a formal state transition system, where a system state is conveyed compositionally, as the union of individual states attributed to instances (in our case), or disjoint volatile components in more generic terms. Thus, a kP system state S is an aggregate of S_C, the set of compartment states and μ which denotes the membrane structure as a set of interconnections between compartments. A compartment state is identified by its associated multiset configuration at a particular computational step, together with the membrane type the compartment it subject to. The following set like expression exemplifies a kernel P system state for three compartments c_1, c_2 and

c_3, of types t_1, t_2, t_2, having configurations $2a\ b$, $a\ 2c$ and *empty* respectively. The second fragment is a set of pairs which symbolize links between compartments: c_1 is connected to both c_2 and c_3, who do not share a link in-between.

$$(\{(c_1, t_1, \{2a, b\}), (c_2, t_2, \{a, 2c\}), (c_3, t_2, \{\})\}, \{(c_1, c_2), (c_1, c_3)\})$$

Since kP systems feature a dynamic structure by preserving structure changing rules such as membrane division, dissolution and link creation/destruction, a state defined in this expansive context is consequently variable in size. This is not unnatural for a computational model, however it does become an issue when conflicting with the requirement of a fixed sized pre-allocated data model imposed by most model checker tools. The instinctive solution is to bound the expansion of these collections to a certain maximum based on the algorithmic necessities. For instance, an initial analysis of the problem we are modelling can provide relevant details about the number of steps required for a successful execution, the number of divisions that may occur and the maximum number of links generated.

One of the most fruitful advantages of model checking is the fact it can be completely automated. The principal insight is that both the system's state space (commonly referred to as *global reachability graph*) and the correctness claim specified as a temporal logic formula can be converted to non-deterministic finite automata. The product of the two automata is another NDFA whose accepted language is either empty in which case the correctness claim is not satisfied, or non-empty if the system exhibits precisely the behaviour specified by the temporal logic statement. There are numerous implementations of this stratagem boasting various supplementary features, a survey of which is beyond the scope of this study. The model checker extensively adopted in formal verification research on membrane systems is Spin. Developed by Gerard J. Holzmann in the 1990s, Spin is now a leading verification tool used by professional software engineers and has an established authority amidst model checkers. Among plentiful qualities, Spin is particularly suited for modelling concurrent and distributed systems by means of interleaving atomic instructions. For a more comprehensive description of the tool, we refer the reader to [12].

A model checker requires an unambiguous representation of its input model, together with a set of correctness claims expressed as temporal logic formulae. Spin features a high level modelling language, called Promela, which specializes in concise descriptions of concurrent processes and inter-process communication supporting both rendezvous and buffered message passing. Another practical and convenient aspect of the language is the use of discrete primitive data types as in the C programming language. Additionally, custom data types and single dimensional arrays are also supported, although in restricted contexts only.

The kernel P systems specification is an embodiment of elementary components shared by most variants, complemented by innovative new features, promoting a versatile modelling framework without transgressing the *membrane computing* paradigm. Characterised by a rich set of primitives, kP systems offer many high level functional contexts and building blocks such as the exhaustive and arbitrary execution of a set of rules, complex guards and the popular concept of membrane division - powerful modelling instruments from a user centric perspective. An attempt, however, to equate such a complex synthesis of related abstractions to a mainstream specification is a daunting and challenging task. It is perhaps evident that users should be entirely relieved of this responsibility, and all model transformations should be handled automatically. It is precisely this goal which motivates the development of **kpWorkbench**, a basic framework which integrates a set of translation tools that bridge several target specifications we employ for kernel P system models. The pivotal representation medium is, however, the newly introduced kP-Lingua, a language designed to express a kP system coherently and intuitively. kP-Lingua is described in detail in [7], which includes an EBNF grammar of its syntax. We exemplify kP-Lingua in our dedicated case study, presented in section 5 of this paper.

One of the fundamental objectives in devising a conversion strategy is to establish a correspondence with respect to data and functional modules between the two specifications. In some cases, a direct mapping of entities can be identified:

– A **multiset of objects** is encoded as an integer array, where an index denotes the object and the value at that index represents the multiplicity of the object;
– A **compartment type** is translated into a data type definition, a structure consisting of native elements, the multiset of objects and links to other compartments, as well as auxiliary members such as a temporary storage variable, necessary in order to simulate the inherent parallelism of P systems.
– A **compartment** is an instance of a data type definition and a set of compartments is organised into an array of the respective type;
– A **set of rules** is organised according to an **execution strategy** is mapped by a Proctype definition - a Promela process;
– A **guard** is expressed as a composite conditional statement which is evaluated inside an *if* statement;
– A **rule** is generally converted into a pair of instructions which manage subtraction and addition on compartment multisets, but can also process structural elements such as compartments and links;
– **Exhaustive and arbitrary execution** are resolved with using the *do* block;
– **Single non-deterministic execution** is reflected by an *if* statement with multiple branches; we note that Promela evaluates *if* statements differently than most modern programming languages: if more than one branch evaluates to true, then one is non-deterministically chosen.

It is not, however, the simplicity and limpidity of these projections that prevail, especially when dealing with a computational model so often described as unconventional. Rather, concepts such as maximal parallelism and membrane division challenge the mainstream modelling approach of sequential processes and settle on contrived syntheses of clauses. These artificial substitutes operate as auxiliary functions and therefore require abstraction from the global state space generated by a model checker tool. Spin supports the hiding of mediator instruction sets by enveloping code into *atomic* or *d_step* blocks. Although this is a very effective optimisation, we are still faced with the problem of instruction interleaving, the de facto procedure which reconciles parallel and sequential computation. It is not this forced simulation of parallelism that obstructs a natural course for P system verification with Spin, but rather the inevitable inclusion of states generated by interleaved atomic instructions or ensembles of instructions.

In our approach we overcome this obstacle with a hybrid solution, involving both the model in question and the postulated properties. Firstly, 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. The technique is demonstrated in our case study of section 5.

While the approach is a practical success, its efficacy is still a questionable matter. Although a substantial set of states is virtually *neglected* when asserting a correctness claim, the complete state space is nevertheless generated (i.e. including the superfluous states) and each state examined: if the state is flagged as a genuine P system state, then it is queried further, otherwise it is *skipped.* In terms of memory usage, the implications are significant and certainly not to be underestimated, particularly when the model exhibits massively parallel and non-deterministic behaviour.

We conclude this section with an informal synopsis of the *kP system - Promela* translation strategy and the rationale behind some of its noteworthy particularities:

– While each compartment type is represented by a Promela process definition, a *Scheduler* process is employed to launch and coordinate the asynchronous execution of procedures per compartment. The following pseudo-code illustrates the managerial role played by our scheduler:

```
process Scheduler {
    while system is not halted {
        for each type T_i {
            for each compartment C_j of type T_i {
                appoint process P(T_i) to compartment C_j;
            }
        }
        start all appointed processes;
        wait until all appointed processes finish;
        state = step_complete;
        print configuration;
        state = running;
    }
}
```

– Each compartment consists of two multisets of objects, one which rules operate on and *consume* objects from; and the second which temporarily stores the produced or communicated objects. Before the end of each computational step, the content of the auxiliary multiset is committed to the primary multiset, which also denotes the compartment's configuration. This interplay is required to simulate a parallel execution of the system.

4 Queries on kernel P systems

A much debated aspect of model checking based formal verification is specifying and formulating a set of properties whose correctness is to be asserted. Since model checking is essentially an exhaustive state space search, there is a persistent and irreconcilable concern over the limitations of this method when investigating the behaviour of concurrent models, generative of an astronomical state spaces. More precisely, the complexity of the model itself has a great subversive impact on the property gamut which can be employed such that the procedure remains feasible given reasonable computational resources.

It is not just the inherent limitations of this technique which must be taken into consideration, but also the effort and tenacity required to formally express specific queries concisely and faithfully into prescribed logical frameworks. Amir Pnueli's seminal work on temporal logic [20] was a major advance in this direction, enabling the elegant representation of time dependent properties in deductive systems. Essential adverbial indicators such as *never* and *eventually* have a diametric correspondent in temporal logic, as operators which relate system states in terms of reachability, persistence and precedence, supporting more powerful queries in addition to simple state equivalence assertions and basic invariance. Exploiting the potential of these logics, as evident as it may seem, can still be problematic and laborious under certain circumstances.

Firstly, devising a temporal logic formula for a required property is a cumbersome and error-prone process even for the experienced. It is often the case that the yielded expressions, although logically valid, are counter-intuitive and abstruse, having little to tell about the significance of the property itself. As with any abstraction that is based on pure logical inference, it is devoid of meaning outside the logical context. To clearly emphasize our affirmations, consider the following example:

$$\begin{aligned}&\mathrm{G}\ (vm\_functional = true\ \wedge\ vm\_coin > 0 \rightarrow\\ &\quad \mathrm{F}\ (vm\_dipsensed\_drink > 0\ \wedge\ \mathrm{F}\ (vm\_coin = 0)) \vee\\ &\qquad \mathrm{F}\ (vm\_functional = false))\end{aligned}$$

is a faithful LTL (linear time temporal logic) representation of a property which can be phrased as *"a vending machine, if functional, will always dispense a drink after having accepted coins and will either become dysfunctional or its coin buffer will be depleted."* Although we have used intuitive variable names, it is not immediately apparent what this expression stands for, requiring a thorough understanding of the LTL specification together with effort and insight to accurately decipher its meaning.

The second notable issue we wish to evince is the correctness of the formula itself which can often be questionable even if the property is of moderate complexity and is syntactically accepted by a model checker tool. How can one prove that a temporal logic expression is indeed a valid representation of a property we wish to verify? Is this a genuine concern we should address, or is it acceptable to assume the faithfulness of temporal logic expressions to specific queries, as formulated by expert and non-expert users?

In response to these controversies, we propose a strategy that facilitates a *guided construction* of relevant LTL properties and automates the translation to their formal equivalent. It is the Natural Language Query (NLQ) builder that was developed to support this methodology. The tool features a rich set of *natural language patterns*, presented to users as sequences of GUI (graphical user interface) form elements: labels, text boxes and drop-down lists. Once the required values have been selected or directly specified and the template populated, NLQ automatically converts the natural language statement to its temporal logic correspondent. The translation from an informal to a formal representation is based on an interpreted grammar which accompanies each natural language pattern.

In table 1, we illustrate a selection of patterns whose instantiation generates properties suitable for kP system models and their formal verification. Table 2 depicts the EBNF based grammar according to which, *state formulae* are derived, with reference to kernel P system components.

In order to verify kP systems modeled in kP-Lingua using Spin model checker, properties specified in LTL should be reformulated in Spin language for the corresponding Promela model. In Table 3, we give LTL formulae of the patterns shown in Table 1, and their corresponding translations in Spin language for the Promela specification. Each LTL formula described for P systems in general (and kP systems in our case) 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 computation step is completed) or represents an intermediate state (it is false if intermediary steps are executed) [15, 18].

The idea of capturing recurring properties into categories of patterns was initiated by Dwyer et al. in their seminal paper of 1999 [6]. This study surveyed more than five hundred temporal properties and established a handful of pattern classes. In [9], this mapping was extended to included additional time related

Pattern	::= Occurrence | Order
Occurrence	::= Next | Existence | Absence | Universality | Recurrence | Steady-State
Order	::= Until | Precedence | Response
Next	::= stateFormula *'will hold in the next state'.*
Existence	::= stateFormula *'will eventually hold'.*
Absence	::= stateFormula *'never holds'.*
Universality	::= stateFormula *'always holds'.*
Recurrence	::= stateFormula *'holds infinitely often'.*
Steady-State	::= stateFormula *'will hold in the long run (steady state)'.*
Until	::= stateFormula *'will eventually hold, until then'* stateFormula *'holds continuously'.*
Response	::= stateFormula *'is always followed by'* stateFormula.
Precedence	::= stateFormula *'is always preceded by'* stateFormula.

Table 1. Grammar for query patterns.

patterns and their associated observer automata. This was further supplemented with real-time specification patterns in [16].

A unified pattern system was introduced in [2], adding new real-time property classes. Probabilistic properties were similarly catalogued based on a survey of 200 properties [10], and provisioned with a corresponding structured grammar.

An analogous undertaking can also be observed in [19], where an array of query templates which target biological models was proposed.

Although the NLQ builder is based on an extensive set of patterns investigated in above mentioned literature, the templates relevant to our formal examination of kP system models represent a small subset of this collection; particularly we only employ patterns which generate temporal properties.

5 Case study: the Subset Sum problem

In this section we demonstrate the proposed methodology with a case study, the subject of which is the well known Subset Sum problem.

The Subset Sum problem is stated as follows:

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 $k \in N$, 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 Subset Sum problem is representative for the NP complete class because it portrays the underlying necessity to consider *all combinations* of distinct elements of a finite set, in order to produce a result. Consequently, such a problem requires exponential computational resources (assuming P $\neq$ NP), either in the

stateFormula	::=	statePredicate | statePredicate *‘does not hold’* | stateFormula *‘and’* stateFormula | stateFormula *‘or’* stateFormula
statePredicate	::=	numericExpression relationalOperator numericExpression
numericExpression	::=	objectCount | localObjectCount | compartmentCount | linkCount | linkToCount | numericLiteral
linkCount	::=	*‘the number of links from’* compartmentQuery *‘to’* compartmentQuery
linkToCount	::=	*‘the number of links to’* compartmentQuery
compartmentQuery	::=	*‘all compartments’* | *‘compartments’* compartmentCondition
compartmentCondition	::=	*‘of type’* typeLabel | *‘of type other than’* typeLabel | *‘linked to’* compartmentQuery | *‘not linked to’* compartmentQuery | localObjectCount relationalOperator numericExpression | linkToCount relationalOperator numericExpression
localObjectCount	::=	*‘the number of objects’* localObjectCondition
objectCount	::=	*‘the number of objects’* objectCondition
localObjectCondition	::=	*‘with label’* objectLabel | *‘with label different than’* objectLabel | localObjectCondition *‘and’* localObjectCondition | localObjectCondition *‘or’* localObjectCondition
objectCondition	::=	localObjectCondition | *‘in’* compartmentQuery | *‘not in’* compartmentQuery | objectCondition *‘and’* objectCondition | objectCondition *‘or’* objectCondition
relationalOperator	::=	*‘is equal to’* | *‘is not equal to’* | *‘is greater than’* | *‘is less than’* | *‘is greater than or equal to’* | *‘is less than or equal to’*
numericLiteral	::=	? {0-9} ?

Table 2. EBNF based grammar for state formulae.

Pattern	**Informal Formula**	**LTL formula**	**Spin formula**
Next	p will hold in the next state	X p	`X(!pInS U (p && pInS))`
Existence	p will eventually hold	F p	`<>(p && pInS)`
Absence	p never holds	$\neg$(F p)	`!(<>(p && pInS))`
Universality	p always holds	G p	`[] (p || !pInS)`
Recurrence	p holds infinitely often	G F p	`[](<>(p && pInS) || !pInS)`
Steady-State	p will hold in the steady state	F G p	`<>([](p || !pInS) && pInS)`
Until	p will eventually hold, until then q holds continuously	p U q	`(p || !pInS) U (q && pInS)`
Response	p is always followed by q	G ($p \rightarrow$ F q)	`[]((p -> <> (q && pInS)) || !pInS))`
Precedence	p is always preceded by q	$\neg(\neg p$ U $(\neg p \wedge q))$	`!((!p || !pInS) U (!p && q && pInS))`

Table 3. LTL formulae and translated Spin specifications of the property patterns

temporal (number of computational steps) or spatial (memory) domain, or both. The Subset Sum problem explicitly denominates combinations of integers as subsets of the initial set A, or more accurately, the set of weights respective to A. It is therefore transparent that the number of all combinations which can be generated and evaluated is the cardinality of the power set of A, that is 2^n. Since our elements are in fact integers, optimisations have been considered, leveraging the intrinsic order relation between numbers, coupled with efficient sorting algorithms to avoid generating all possible subsets [13]. This did not, however, manage to reduce the complexity of the problem to a non-exponential order.

P system variants endowed with *membrane division* proved to be ideal computational frameworks for solving NP complete problems efficiently. The insightful strategy, often referred to as *trading space for time*, can be envisaged as the linear generation of an exponential computational space (compartments) together with the linear distribution (replication) of constituent data (multiset of objects). The topic is very popular in the community and was subject to extensive investigation; while the underlying principle is pertinent to our study, we shall illustrate it more sharply as applied, using a kernel P system model to solve the Subset Sum problem:

Consider the kP system

$$k\Pi = (\{a, x, step, yes, no, halt, r_1, \dots, r_n\}, \mu, (Main, \{a\}), (Output, \{step\}))$$

with μ represented by a link between the two instances of type *Main* and *Output* respectively.

The rules for compartments of type *Main* are:

- R_i: $a \longrightarrow [a, r_i][w_i x, a, r_i]\{\neg r_i\}$, $1 \le i \le n$
- R_{n+1}: $a \longrightarrow (yes, halt)_{Output}$ $\{= kx\}$
- R_{n+2}: $a \longrightarrow \lambda$ $\{> kx\}$

where

- n is the number of elements in set A, that is the cardinal of A;
- r_i with $1 \le i \le n$ is an object which flags the execution of a membrane division rule, prohibiting multiple applications of the same addition;

– w_i is the weight of the ith element in the set A, with $1 \leq i \leq n$;
– k is the constant we refer to, when assessing the sum of the values in a subset; if $\sum_{w_i} = k$, then a solution has been found;

The execution strategy $\sigma(Main)$ unfolds as follows:

$$\sigma(Main) = (R_{n+1}, R_{n+2})\&(R_{1..n})$$

Thus, each step a compartment of type *Main* performs two preliminary evaluations: if the number of x objects is precisely k, then a *yes* and a *halt* object are sent to the output membrane. We recall the specialised *halt* object as a universal, model independent and convenient means of halting a computation for kernel P systems: when such an object is encountered in any of the system's compartments, the execution stops at the completion of the computational step. This is generally preferred to specifying halting conditions which relate to configurations or system states particular to the modelled problem.

If the multiplicity of x is greater than k, a condition assessed with the guard $> kx$, the compartment is dissolved, pruning a fruitless search path. Otherwise, a division rule is selected non-deterministically, splitting the compartment in two and adding $w_i x$s to the current multiplicity of x in one of the newly created regions, while preserving the weight of x in the other. Both compartments also receive a r_i object which marks the execution of the ith rule. This will be prevented from executing a second time by the guard $\neg r_i$. The object a is auxiliary and recurs in every compartment of type *Main*.

There is only one compartment of type *Output* which persists throughout the execution, playing the role of an output membrane, as its name plainly indicates: either it receives a *yes* object if a solution is found, or it generates a *no* object if the computation does not halt after $n + 1$ steps. The two rules which correlate with this behaviour are:

– $R_1 : \quad step \longrightarrow 2step$
– $R_2 : (n + 2)step \longrightarrow no, halt$

The rules are executed sequentially:

$$\sigma(Output) = (R_1 \& R_2)$$

Remark 1. The illustrated algorithm is a faithful **linear time** solution to the Subset Sum problem: it computes an answer to the stipulated enquiry in **maximum $n + 2$ steps**, where n is the cardinality of the set A of elements.

Remark 2. The algorithm will generate the sums of all subsets of A in linear time using membrane division; the process is interrupted when a solution is found and computation halts at this stage. A notable difference to the skP (simple kernel P) system based solution presented in [14], is the use of non-deterministic choice in the selection of division rules. This rather unconventional approach facilitates the generation of subset sums that is irrespective of the order of elements in A. Evidently, the artifice owes its merit to the commutativity of integer addition.

Remark 3. The kP system model requires a total of: $n+6$ distinct objects, $n+4$ rules of which $n+1$ employ basic guards and a maximum of 2^n+1 compartments.

Remark 4. Although we have extensively referred to integer weights (of the elements in A) throughout this section, it is important to note that we can not directly represent negative numbers as object multiplicities alone (some encoding can be devised for this purpose). Since the only mathematical operation required is addition, which is a monotonically increasing function, a simple translation to the positive domain can be mapped on the set of weights $w(A)$, which in turn makes this issue irrelevant.

We next demonstrate the implementation of our kP system model in kP-Lingua, highlighting some of the most prominent features of its syntax. The illustrated model maps an instance of the Subset Sum problem with $n = 7$ elements: $w(A) = \{3, 25, 8, 23, 5, 14, 30\}$ 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 -> 2 step .
    9 step -> no, halt .
}

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

The code comprises of two type definitions, *Main* and *Output*, together with the instantiation of two, linked, compartments of the respective types. The first two rules are guarded by $\{= 55x\}$ and $\{> 55x\}$ respectively, and organized in a *choice* block since they are mutually exclusive and each may execute once and only once. Indeed, enclosing these rules in a maximally parallel grouping would result in equivalent behaviour. A guard always relates to the multiset contained in the compartment it evaluates in and terminates with a colon; the $- >$ symbol denotes the transition of a non-empty multiset on the left hand side to a rewrite-communication outcome (objects *yes*, *halt* into the compartment of type Output), or a *single* structure changing element (# which symbolises membrane dissolution). Next, the choice block is applied as a non-deterministic selection of *one* of the rules it envelopes: there are seven division rules, which resemble the addition of a value from $w(A)$. Each rule is prefixed by a guard $!r_i$, in order to prevent its subsequent application which would equate to multiple additions of the same number.

Type *Output* lists two rewriting rules which execute successively and non-repetitively. The first rule *increments* the number of *step* objects in the compartment, updating the step count as the computation unfolds. The second rule will only execute if we have reached the 9th step and no *halt* object was received from any of the *Main* compartments, effectively pronouncing a negative answer to the problem.

The kP-Lingua implementation is a compact and intuitive representation of the formally described model presented earlier. The specification is next translated into Spin's modelling language, Promela, a fully automated process accomplished by a *kP-Lingua parser* and *kP system - Promela* model converter, constituent tools of *kpWorkbench*. We document this stage of our approach with several fragments of the rather cryptic Promela encoding, as generated by our converter.

```
#define A0_SIZE 9
#define A1_SIZE 4

typedef C0 {
    int x[A0_SIZE] = 0;
    int xt[A0_SIZE] = 0;
    int c1Links[1];
    int c1LCount = 0;
    int c1LSize = 0;
    bit isComputing = 0;
    bit isDissolved = 0;
    bit isDivided = 0;
}

typedef C1 {
    int x[A1_SIZE] = 0;
    int xt[A1_SIZE] = 0;
    int c0Links[100];
    int c0LCount = 0;
    int c0LSize = 0;
    bit isComputing = 0;
}

int step = 0;
bit halt = 0;
```

```
C0 m0[20];
int m0Count = 0;
int m0Size = 0;
C1 m1[1];
int m1Count = 0;
int m1Size = 0;

int m0DissolvedCount = 0;
int stepsExp = 1;
```

In table 4 we elucidate the constituent elements of the above printed data structures and variable declarations.

A0_SIZE, A1_SIZE	The size of the alphabet for each type of compartment;
C0, C1	The compartment types *Main* and *Output* respectively;
x, xt	The arrays which store multiplicities of objects encoded as indices;
c1Links[1]	The array of links to compartments of type C1;
isComputing	A flag indicating whether a process is running on this instance or not;
isDissolved	A flag indicating whether the compartment is dissolved or not;
isDivided	Indicates if the compartment was divided (and henceforth considered non-existent);
m0, m1	The arrays which store compartments of type C0 (*Main*) and C1 (*Output*), respectively;
m0[0].x[2]	The object with index 2 in the 0th compartment of type C0;
m1[0].x[0]	Multiplicity of object *step* in compartment 0 of type *Output*;
m1[0].x[1]	Multiplicity of object *yes* in the output compartment;
m1[0].x[2]	Multiplicity of object *no* in the output compartment;
m1[0].x[3]	Multiplicity of object *halt* in the output compartment;
m0DissolvedCount	The number of dissolved compartments of type *Main*;
stepsExp	A number updated each step with the value of 2^{step}.

Table 4. Interpretation of variable expressions generated in Promela

The second key requirement for the model checking methodology we exemplify in this section is the provision of LTL formulae the validity of which is to be asserted against the model. As methodically described in the previous section, a set of properties is generated by instantiating various natural language patterns. These are appointed as templates to be completed by the user with model variables or numeric constants, interactively, through a graphical user interface. Several screenshots which illustrate the Natural Language Query (NLQ) builder, integrated into kpWorkbench are supplied in the Appendix.

Table 5 lists an array of ten properties we have compiled and derived from natural language patterns for the Subset Sum example. These properties have been successfully verified with Spin on a Core i7 980X based machine, with 24GB RAM and running Windows 8 Professional Edition.

Devising a set of properties assisted by the NLQ tool becomes an intuitive, effortless and streamlined task, however, there may be cases when a generated

Property	Pattern	Natural Language Statement and Spin formula

1	Until	The computation will eventually halt.
		`halt == 0 U halt > 0`
		`(m1[0].x[3] == 0 || !pInS) U (m1[0].x[3] > 0 && pInS)`

2	Until	The computation will halt within $n + 2$ steps.
		`(halt == 0 && steps < n + 2) U (halt > 0 && steps <= n + 2)`
		`(m1[0].x[3] == 0 && m1[0].x[0] < n+2 || !pInS) U`
		`(m1[0].x[3] >= 0 &&m1[0].x[0] <= n+2 && pInS)`

3	Until	The computation will eventually halt with either a ‘yes’ or ‘no’ result.
		`halt == 0 U (halt > 0 && (yes > 0 || no > 0))`
		`(m1[0].x[3] == 0 || !pIns) U`
		`(m1[0].x[3] > 0 && (m1[0].x[1] > 0 || m1[0].x[2] > 0) && pInS)`

4	Until	At least one membrane division will eventually take place (before a result is obtained).
		`(yes == 0 && no == 0) U m0Count > 1`
		`(m1[0].x[1] == 0 && m1[0].x[2] == 0) || !pInS U m0Count > 1 && pInS`

5	Existence	A ‘yes’ result is eventually observed within no more than three steps.
		`F (yes > 0 && steps <= 3)`
		`<> (m1[0].x[1] > 0 && m1[0].x[0] <= 3 && pInS)`

6	Existence	A ‘yes’ result is eventually observed within more than three steps.
		`F (yes > 0 && steps > 3)`
		`<> (m1[0].x[1] > 0 && m1[0].x[0] > 3 && pInS)`

7	Existence	A result (‘yes’ or ‘no’) is eventually obtained without any membrane dissolutions.
		`F (yes > 0 || no > 0) && m0DissolvedCount == 0`
		`<> ((m1[0].x[1] > 0 || m1[0].x[2] > 0) && m0DissolvedCount == 0 && pInS)`

8	Existence	A ‘yes’ result is eventually obtained with membrane dissolution occuring.
		`F yes > 0 && m0DissolvedCount > 0`
		`<> (m1[0].x[1] > 0 && m0DissolvedCount > 0 && pInS)`

9	Universality	The number of compartments in use is always equal to $2^{stepcount}$.
		`G m0Count + 1 == TwoToTheNumberOfSteps`
		`[] (m0Count + 1 == TwoToTheNumberOfSteps || !pInS)`

10	Absence	There will never be a negative answer for this example.
		`!F no > 0`
		`!(<> (m1.x[2] > 0 && pInS ))`

Table 5. List of properties derived from natural language patterns using NLQ and their generated LTL equivalent

natural language statement does not reflect the meaning of the property in its entirety, although it is logically equivalent. This may lead to shallow interpretations if the formal representation is not consulted and ultimately to oversights of relevant implications of the property. For example, in Table 5, the property *a 'yes' result is eventually observed within no more than three step* is as a fabricated form of *there exists a non-deterministic execution strategy that yields an affirmative result to the problem in no more than three steps.* The second expression is significantly more elevate and meaningful in comparison with its generated counterpart which clearly describes the underlying LTL formulae, but requires a deeper understanding of the model for an accurate interpretation.

6 Conclusions

The approach to kernel P system model checking presented in this paper is a powerful synthesis of concepts and ideas, materialised into an aggregate of software tools and template data sets. The investigation permeates two innovative leaps, namely the kP system computational model in the context of membrane computing and the use of natural language patterns to generate temporal logic properties in the field of model checking. After establishing a model equivalence relation together with a procedural translation from a generic representation to a notation required by Spin, non-specialist users can benefit from the standard features offered by the model checker. The often intricate and abstruse process of constructing temporal logic formulae has also been abstracted to natural language statements and interactive visual representation through graphical user interface (GUI) elements. Another consequential advantage of significance is the correctness guarantee conferred by an automatic model conversion and formula generation.

Our case study illustrated in section 5, demonstrates the feasibility of this approach with its illustrious qualities, but also exposes the potential limitations of the method: on one hand, the notorious state space explosion problem is an inexorable fact that circumscribes the model checking of concurrent and non-deterministic systems; on the other hand, some generated properties, products of composite natural language patterns, are devoid of meaning and can possibly lead to shallow or inaccurate interpretations and even confusion.

Evidently, a more consistent qualitative evaluation of the methodology, involving several other case studies is required to highlight its potential and limitations more generally. It would be interesting to see the outcome of future investigations in this newly established context.

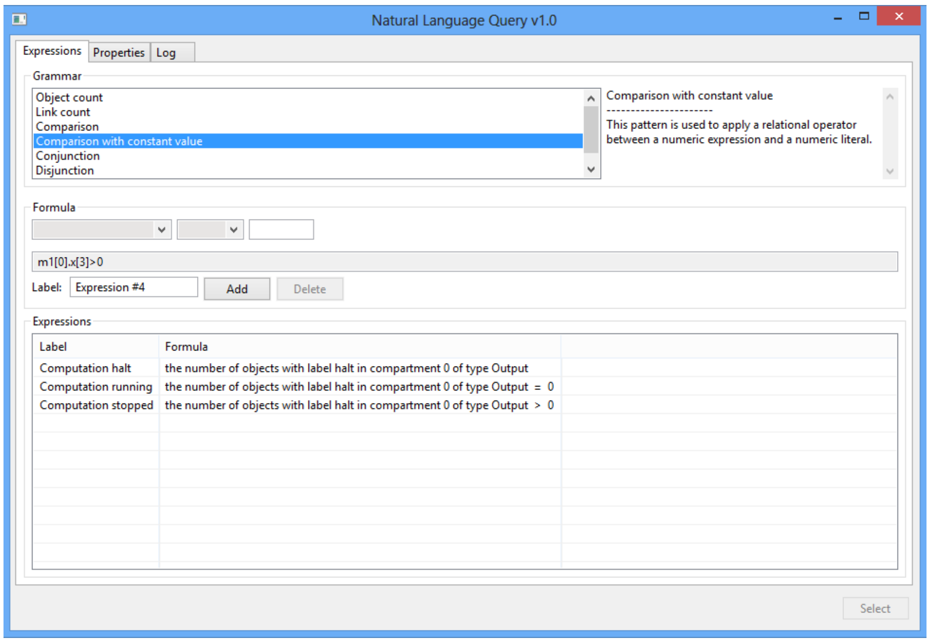
Acknowledgements. The work of FI, RL and LM was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). SK acknowledges the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1).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

References

1. Andrei, O., Ciobanu, G., Lucanu, D.: A rewriting logic framework for operational semantics of membrane systems. Theoretical Computer Science 373(3), 163–181 (2007)
2. Bellini, P., Nesi, P., Rogai, D.: Expressing and organizing real-time specification patterns via temporal logics. J. Syst. Softw. 82(2), 183–196 (Feb 2009)
3. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: WMC 2007. LNCS, vol. 4860, pp. 138–159. Springer (2007)
4. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics workbench - A P systems based tool for systems and synthetic biology. Applications of Membrane Computing in Systems and Synthetic Biology. To Appear (2013)
5. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Natural Computing Series, Springer (2006)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering. pp. 411–420. ICSE '99, ACM (1999)
7. Gheorghe, M., Ipate, F., Dragomir, C., Mierla, L., Valencia-Cabrera, L., Garcia-Quismondo, M., Perez-Jimenez, M.: Kernel P systems. Eleventh Brainstorming Week on Membrane Computing pp. 97–124 (2013)
8. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M.J., Turcanu, A., Valencia-Cabrera, L., García-Quismondo, M., Mierla, L.: 3-Col problem modelling using simple kernel P systems. International Journal of Computer Mathematics 90(4), 816–830 (2013)
9. Gruhn, V., Laue, R.: Patterns for timed property specifications. Electron. Notes Theor. Comput. Sci. 153(2), 117–133 (2006)
10. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proceedings of the 30th international conference on Software engineering. pp. 31–40. ICSE '08, ACM (2008)
11. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Proc. TACAS'06. LNCS, vol. 3920, pp. 441–444. Springer (2006)
12. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 275–295 (1997)
13. Horowitz, E.; Sahni, S.: Computing partitions with applications to the knapsack problem. Journal of the Association for Computing Machinery 21, 277–292 (1974)
14. Ipate, F., Lefticaru, R., Mierla, L., Cabrera Valencia, L., Han, H., Zhang, G., Dragomir, C., Perez Jimenez, M., Gheorghe, M.: Kernel P systems: Applications and implementations. In: Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013, Advances in Intelligent Systems and Computing, vol. 212, pp. 1081–1089. Springer Berlin Heidelberg (2013)
15. 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)
16. Konrad, S., Cheng, B.: Real-time specification patterns. In: Proceedings of 27th International Conference on Software Engineering. pp. 372 – 381 (2005)

17. Lefticaru, R., Ipate, F., Valencia-Cabrera, L., Turcanu, A., Tudose, C., Gheorghe, M., Jiménez, M.J.P., Niculescu, I.M., Dragomir, C.: Towards an integrated approach for model simulation, property extraction and verification of P systems. Tenth Brainstorming Week on Membrane Computing vol. I, 291–318 (2012)
18. Lefticaru, R., Tudose, C., Ipate, F.: Towards automated verification of P systems using Spin. International Journal of Natural Computing Research 2(3), 1–12 (2011)
19. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. Bioinformatics 24(16), i227–i233 (2008)
20. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society Press (1977)
21. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)

Appendix A: Screenshots illustrating our NLQ tool



Object count
Link count
Comparison
Comparison with constant value
Conjunction
Disjunction

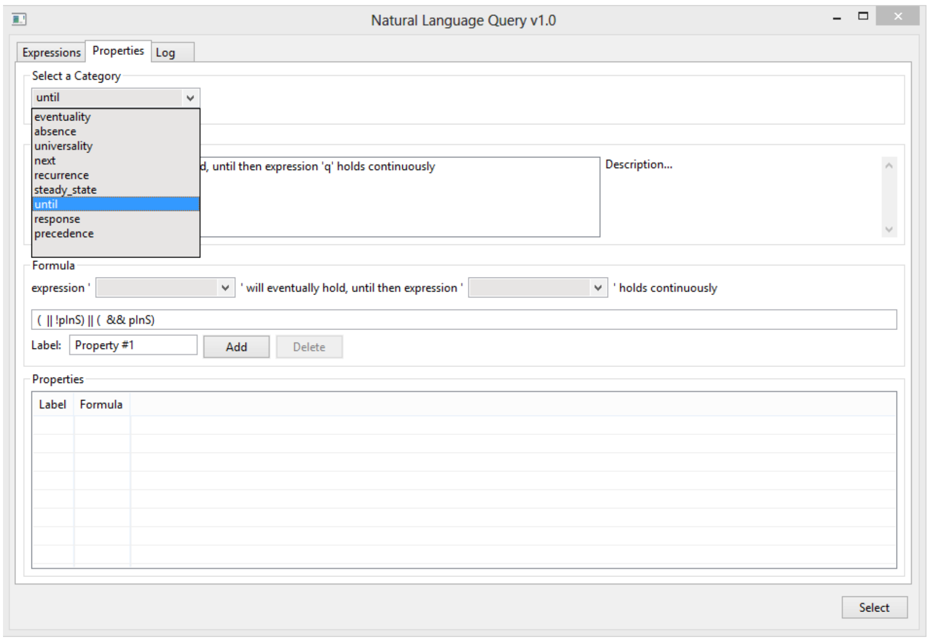
Comparison with constant value

This pattern is used to apply a relational operator between a numeric expression and a numeric literal.

m1[0].x[3]>0

Label: Expression #4

Label	Formula
Computation halt	the number of objects with label halt in compartment 0 of type Output
Computation running	the number of objects with label halt in compartment 0 of type Output = 0
Computation stopped	the number of objects with label halt in compartment 0 of type Output > 0



Natural Language Query v1.0

Expressions | Properties | Log

Select a Category

until

eventuality
absence
universality
next
recurrence
steady_state
until
response
precedence

d, until then expression 'q' holds continuously

Description...

Formula

expression ' ' will eventually hold, until then expression ' ' holds continuously

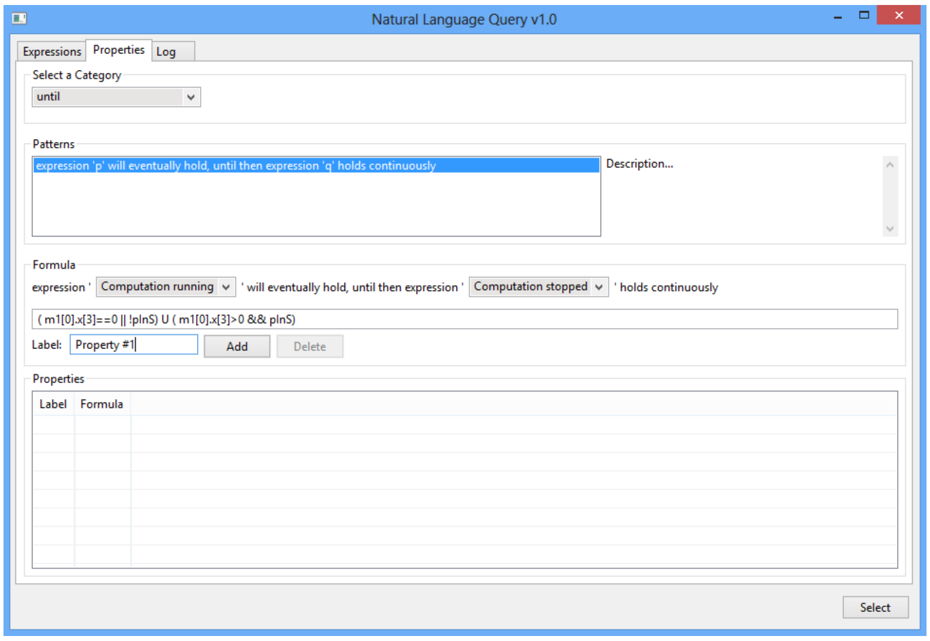
(|| !pInS) || (&& pInS)

Label: Property #1 Add Delete

Properties

Label	Formula

Select



Natural Language Query v1.0

Expressions | Properties | Log

Select a Category

until

Patterns

expression 'p' will eventually hold, until then expression 'q' holds continuously

Description...

Formula

expression ' Computation running ' will eventually hold, until then expression ' Computation stopped ' holds continuously

(m1[0].x[3]==0 || !pInS) U (m1[0].x[3]>0 && pInS)

Label: Property #1 Add Delete

Properties

Label	Formula

Select