Journal of Membrane Computing (2026) 8:315–329
https://doi.org/10.1007/s41965-026-00221-7

REVIEW PAPER

Some new developments in kP-Lingua

Ştefan Iacob[1] · Florentin Ipate[1] · Ionuţ Mihai Niculescu[1] · Savas Konur[2]

Received: 23 December 2025 / Accepted: 28 March 2026 / Published online: 11 June 2026
© The Author(s) 2026

Abstract
In this paper, we introduce a new indexation mechanism within a general class of P system models called kernel P systems, together with kPWorkbench, a software framework developed to support them. Our approach enables a compact definition of general patterns of rules, guards, objects, and compartments using syntactic constructs that are automatically expanded into fully specified kernel P systems. We also illustrate our approach through a case study, the Partition Problem.

Keywords Membrane systems · kernel P systems · Model checking

1 Introduction

Membrane computing is a bio-inspired computing paradigm introduced by Gh. Păun [35]. It uses models called *membrane systems* or *P systems*, which are inspired by the structure and functionality of the living cell. Membrane computing research has evolved very rapidly and consistently, and it covers both theoretical aspects and a significant number of applications, with numerous variants of membrane systems being proposed and investigated. A first research monograph was published two years after the field emerged [30], and then a comprehensive handbook [32] and a survey paper [38] appeared. Applications in systems and synthetic biology [22] and real-life complex problems have also been investigated with this computing paradigm [10, 43]. The primary limitation of this bio-inspired model lies in the absence of physical implementations. Unlike DNA computing, current technology does not enable the realization of P systems in laboratory settings, in vitro, in vivo, or through any electronic media. Membrane systems have been extensively explored from a theoretical standpoint [26, 28], and have also been applied to modeling complex systems across diverse fields, including computational biology [16, 17, 31] and robotics [4, 29, 36]. In the absence of physical implementations, researchers have turned to software tools capable of simulating P system computations. These tools provide a virtual environment to apply and test the implementations based on P systems, offering insights into their computational behavior. Simulators are crucial resources for P system design and validation. Researchers have developed several simulators to date, which can simulate some of the proposed P system models. As membrane computing continues to evolve, new variants of P system models are introduced to address emerging theoretical challenges and application domains. This dynamic growth has led to the development of numerous specialized simulators tailored to specific model types. For example, simulation tools have been created for tissue P systems with evolutional symport/antiport rules [39], spiking neural P systems [9, 18, 33], and dynamic threshold neural P systems [34]. These simulators extend the applicability of membrane computing and support the exploration of biologically inspired mechanisms in domains, such as neural computation, communication networks, and pattern recognition. Due to the characteristics of P systems, executing them on traditional Von Neumann architecture computers remains inefficient. As a result, efforts have been made to leverage GPUs for parallel emulation [2, 46]. Nonetheless, these simulations aim to replicate similar outcomes rather than accurately reproducing the true internal dynamics of a P system. In addition, in [5], an initial intrinsically parallel hardware architecture for implementing generic P system models is proposed. It is specifically designed for straightforward implementation on programmable logic devices such as FPGAs. However, as membrane computing models grow in size and complexity, so does the effort required to define them. Many rules and structures in P system models fol-

✉ Ştefan Iacob
stefan.iacob@s.unibuc.ro

[1] Faculty of Mathematics and Computer Science, University of Bucharest, 14 Academiei St, 010014 Bucharest, Romania

[2] School of Computing and Engineering, University of Bradford, West Yorkshire, Bradford BD7 1DP, UK

low repetitive patterns, differing only in parameters, such as indices or object names. Manually specifying these instances is error-prone and hampers scalability and maintainability. To address this challenge, we introduce a new indexation mechanism within a general class of P system models called kernel P systems [11], together with kPWorkbench, a software framework developed to support them. Our approach enables a compact definition of general patterns of rules, guards, objects, and compartments using syntactic constructs that are automatically expanded into fully specified kernel P systems. These simulators are crucial resources for P system design and validation (see more details regarding related simulation frameworks in Sect. 6).

This paper presents the indexation mechanism in kPWorkbench, its syntax and semantics, and its impact on model conciseness and expressiveness. We demonstrate the utility of the mechanism through a case study, a kernel P System solution to the NP-complete Partition Problem. We also provide a comparative evaluation with other simulation frameworks and outline future directions for extending the indexation approach. The concept of indexation as a means to increase modeling efficiency is not unique to kPWorkbench. Other simulation frameworks, such as P-Lingua 5 and UPSimulator, have also incorporated mechanisms that permit the definition of indexed or parameterized rules. The paper makes an original contribution to the membrane computing field by introducing an indexation mechanism that significantly enhances the modeling capabilities and usability of P system models. Our approach improves the practicality of membrane computing tools, bridging the gap between theoretical frameworks and real-world applications, and opening new possibilities for tackling complex real-world problems. This paper is organized as follows: Section 2 provides an overview of kPWorkbench and summarizes its main architectural and functional features as a simulation and verification platform for kernel P systems. Section 3 introduces the unified indexation mechanism, detailing its syntax, semantics, and practical use across rules, guards, multisets, constants, and compartment links, together with an illustrative broadcasting model. Section 4 presents the verification framework of kPWorkbench and explains how indexation is seamlessly integrated into property specification and temporal logic checking. Section 5 applies the new indexation capabilities to the Partition Problem, demonstrating both dynamic and static subset generation strategies, and discusses the resulting computational behavior. Section 6 reviews related simulation frameworks and compares their capabilities with those of kPWorkbench, highlighting its unique support for indexed guards, modular specification, and formal verification. Finally, Sect. 7 concludes the paper and outlines planned extensions, including set-based indexation, named vectors, and indexed temporal properties.

2 kPWorkbench: a simulator for kernel P systems

Among various types of membrane systems studied for different purposes, a variant called the *kernel P system* (*kP system*, for short) has been introduced to provide a coherent computational framework where several features available in various membrane system variants can work together in a consistent and efficient manner [12]. These kP systems are specified in a domain-specific language, called *kP-Lingua*, allowing models to be simulated with a software tool called kPWorkbench [1]. Conceptually, a kP system is structured as a network of compartments, each defined by a specific type, containing multisets of objects, and governed by a set of rewriting and communication rules. The application of these rules is tightly controlled by execution strategies, such as maximal parallelism or non-deterministic choice, and can be further constrained by logical guards that evaluate the current multiset configuration of the compartment. This tool's unique features, including modeling, simulation, agent-based high-performance simulation, and verification, enable the comprehensive computational analysis of membrane systems.

A traditional challenge in computational modeling is the steep learning curve associated with formal verification, particularly for interdisciplinary researchers. To address this, kPWorkbench incorporates a user-friendly property language based on natural language statements. This abstraction makes the complex task of property specification significantly simpler for non-experts. The core capabilities of the tool are presented in detail in [24]. Furthermore, the robust capabilities of kPWorkbench, alongside the expressive power and effectiveness of the underlying kP system formalism, have been successfully validated through several representative case studies [15, 21, 23, 25]. Ultimately, as a comprehensive software platform that integrates advanced modeling and analysis techniques, kPWorkbench serves as an essential bridge between the theoretical investigation of kP systems and their practical computational application.

3 Indexation in kPWorkbench

We will further elaborate on the indexation mechanism and its applications, supported by examples. The implementation includes a unified indexation mechanism covering rules, guards, multisets, constants, and compartment links, as well as support for including external files. The use of indexed rules and constant values will be illustrated through a kP-Lingua fragment. The new indexation feature implemented in kPWorkbench facilitates the automatic generation of indexed sequences within kP system models. It uses the `$` symbol to signify the variable to be indexed within an expression,

using the format `rule_pattern: lower_bound <= index <= upper_bound`. For example, an expression of the form `a_$i$ -> [b_$i$, a_$i+1$][a_$i+1$] . : 1 <= i < n` indicates that the variable `i` will be iterated over a specified range. If $n = 6$, the feature generates the following rules:

```
a_1 -> [b_1, a_2][a_2].
a_2 -> [b_2, a_3][a_3].
a_3 -> [b_3, a_4][a_4].
a_4 -> [b_4, a_5][a_5].
a_5 -> [b_5, a_6][a_6].
```

This approach effectively generates a compact rule sequence, where, for each object `a_$i$`, the compartment is divided: one for `a_$i+1$` and `b_$i$`, and another compartment for `a_$i+1$`. It is important to clarify that indexed expressions in kPWorkbench are purely syntactic macros and do not introduce implicit symbols beyond those generated by the specified index ranges. In particular, for an indexed rule of the form `a_$i$ -> [b_$i$, a_$i+1$][a_$i+1$]` with the range $1 \leq i < n$, the expansion produces instances only up to $i = n - 1$, yielding the terminal symbol `a_$n$`, but never referring to `a_$n+1$`. No object `a_$n+1$` is assumed or implicitly defined. As a consequence, the handling of the terminal case must be specified explicitly by a separate rule if required by the model semantics.

The indexation of guards is another new key feature of kPWorkbench.

```
=b_$i$ & =x : v_$i$ -> v . : 1<=i<=n
```

If $n = 3$:

```
=b_1 & =x : v_1 -> v .
=b_2 & =x : v_2 -> v .
=b_3 & =x : v_3 -> v .
```

Guard indexation can also be written as:

```
@operation, expression : range@
```

It is based on the idea of specifying a logical pattern (such as a conjunction or disjunction of expressions) parameterized by an index variable. Here, `operation` can be `&` for conjunction or `|` for disjunction, `expression` is a Boolean condition that may include index variables (denoted by `i`, `j`, etc.), and `range` specifies the bounds of the index variables.

Consider a rule that is guarded by a conjunction and a disjunction of multiple equality tests over indexed variables:

```
(=p0 & =q0) | (=p1 & =q1) | (=p2 & =q2) | (=p3 & =q3) : r -> s .
```

This can be rewritten more compactly using indexation as:

```
@|,(=p$i$ & =q$i$) : 0<=i<=3@ : r -> s .
```

The guard indexation feature is also compatible with rewriting rules indexation, allowing both guards and their associated actions to be indexed independently:

```
(=a0 & =b0) & (=a1 & =b1) | =c : d2 -> e2 .
(=a0 & =b0) & (=a1 & =b1) | =c : d3 -> e3 .
```

This pattern is compactly expressed as:

```
@&,(=a$i$ & =b$i$) : 0<=i<=1@ | =c : d$j$ -> e$j$ : 2<=j<=3 .
```

In kPWorkbench, users can add data to multisets during the initial declaration, without the need to write multiple lines of code.

```
termA {@a_$i$:0<=i<=n@, b, c_7, @d_$j$:0<j<n@ } (Term) .
```

`termA` represents the name or label of the multiset being defined. This serves as an identifier for the multiset and can be used in rules or computations. It defines a family of indexed objects `a_$i$`, where the index `i` ranges from 0 to `n`. The @ symbol is used to distinguish indexed parameters from static elements. If n=4, the resulting output is:

```
$termA {a_0, a_1, a_2, a_3, a_4, b, c_7, d_1, d_2, d_3} (Term) .
```

With the new kPWorkbench version, users can include lines of code from other files using the `#include` directive. The file rewriting_rules.kplt can contain one rule. The file main.kplt can reuse the rules from the file rewriting_rules.kplt.

```
#include "rewriting_rules.kplt"
b -> c .
```

After running the kPWorkbench indexation, the newly generated kpl file will have:

```
a -> b .
b -> c .
```

Another new feature introduced is the indexation between the links of the P system compartments.

```
m1$i$ (L1) - m2$j$ (L2) . : 1<=i<=3, 1<=j<=2
```

After applying the indexation, the result is:

```
m11 (L1) - m21 (L2) .
m12 (L1) - m21 (L2) .
m13 (L1) - m21 (L2) .
m11 (L1) - m22 (L2) .
m12 (L1) - m22 (L2) .
m13 (L1) - m22 (L2) .
```

An indexed expression in kPWorkbench is a syntactic macro that expands into a finite family of standard kP-Lingua constructs. Indexed expressions can occur in rules, guards, multisets, constants, and links.

The unified indexation mechanism described in this section has been implemented as an extension of the kPWorkbench platform. The feature is developed in C# and integrated into kP-Lingua. The complete implementation is publicly available in the kPWorkbench repository on GitHub.

Each indexed expression introduces one or more index variables, which are local to that expression. Thus, two distinct indexation blocks may use the same variable name without conflict.

The grammar of indexed expressions is defined as follows:

⟨*IndexedRule*⟩ ::= ⟨*RuleBody*⟩ : ⟨*RangeList*⟩

⟨*RuleBody*⟩ ::= (. | { | }) ⟨*Expr*⟩*

⟨*IndexedGuard*⟩ ::= @ ⟨*Op*⟩ , ⟨*Condition*⟩ : ⟨*RangeList*⟩ @

⟨*Op*⟩ ::= & | |

⟨*IndexedMultiset*⟩ ::= ID { (⟨*StaticElem*⟩ | ⟨*IndexedElem*⟩)* } (ID) .

⟨*IndexedElem*⟩ ::= @ ⟨*Expr*⟩ : ⟨*RangeList*⟩ @

⟨*IndexedLink*⟩ ::= ID $ ⟨*VarExpr*⟩ $ - ID $ ⟨*VarExpr*⟩ $: ⟨*RangeList*⟩

⟨*RangeList*⟩ ::= ⟨*Range*⟩ (, ⟨*Range*⟩)*

⟨*Range*⟩ ::= ⟨*LowerBound*⟩ <= ⟨*Var*⟩ <= ⟨*UpperBound*⟩ | ⟨*LowerBound*⟩ < ⟨*Var*⟩ < ⟨*UpperBound*⟩ | ⟨*Condition*⟩

⟨*Condition*⟩ ::= ⟨*Expr*⟩ ⟨*RelOp*⟩ ⟨*Expr*⟩

⟨*Expr*⟩ ::= ⟨*Expr*⟩ + ⟨*Expr*⟩ | ⟨*Expr*⟩ - ⟨*Expr*⟩ | ⟨*Expr*⟩ * ⟨*Expr*⟩ | ⟨*Expr*⟩ / ⟨*Expr*⟩ | ⟨*Var*⟩ | NUM | (⟨*Expr*⟩)

⟨*VarExpr*⟩ ::= ⟨*Var*⟩ | ⟨*Expr*⟩

⟨*Var*⟩ ::= [a-z]+ | [0-9]+

⟨*RelOp*⟩ ::= < | <= | > | >= | = | !=

To ensure the correct interpretation of complex guard expressions involving mixed arithmetic, relational, and logical operators, kPWorkbench enforces a strict operator precedence. The precedence levels, ordered from highest (binding tightest) to lowest, are defined in Table 1.

In this subsection, we will present how the indexation features solve a simple broadcasting problem.

3.1 Modeling a broadcasting problem using indexation

To demonstrate the capabilities of the unified indexation mechanism, we first formalize a Broadcasting Problem using the kernel P system notation. This mathematical abstraction defines the general behavior of the system, which will subsequently be instantiated in kP-Lingua. Let the broadcasting system be defined as a kernel P system composed of a central controller compartment of type T_1 and k satellite compartments of type T_2. The system relies on a threshold value, θ, to control the dissemination of messages.

The system is defined as $\Pi = (A, \mu, C_1, \ldots, C_{k+1}, R, i_0)$, where:

- A is the alphabet containing objects $\{a_i, b_i, c, x_i\}$ for $1 \leq i \leq k$.
- The structure μ connects the central compartment C_1 (type T_1) to each satellite compartment C_{j+1} (type T_2), for $1 \leq j \leq k$.
- The initial configuration places objects $\{b_1, \ldots, b_k\}$ in C_1 and a marker $\{x_i\}$ in each C_{i+1}.

The set of rules R is divided into two types, corresponding to the controller and the receivers:

1. Type T_1 (Controller): The controller manages the broadcasting of objects b_i. It operates based on the accumulation of acknowledgement objects c relative to the threshold θ.

- *Broadcasting Rule:* While the number of c objects is below the threshold, b_i is transformed into a_i and sent to the connected T_2 compartment.

$$r_{1,i} : b_i \rightarrow a_i(T_2) \{< \theta c\} \tag{1}$$

- *Termination Rule:* Once the threshold is reached or exceeded, the broadcasting ceases, and b_i is consumed.

$$r_{2,i} : b_i \rightarrow \lambda \{\geq \theta c\} \tag{2}$$

2. Type T_2 (Receiver): The receiver compartments process the incoming message a_i.

- *Acknowledgement Rule:* Upon receiving a_i, the compartment returns b_i to T_1 and generates i copies of the object c. This represents a weighted acknowledgement.

$$r_{3,i} : a_i \rightarrow b_i(T_1)c^i \tag{3}$$

- *Conditional Rule:* A variation of the acknowledgement allows processing only if a specific marker x_i is present.

$$r_{4,i} : a_i \rightarrow b_i(T_1)c^i \{\geq x_i\} \tag{4}$$

Having defined the formal model, we now present the concrete implementation in kPWorkbench. We instantiate the system with specific parameters: the number of indexed families $k = 3$ and the control threshold $\theta = 10$.

The kP-Lingua code below utilizes indexation to compactly represent the families of rules $r_{1,i}$ through $r_{4,i}$, defined

Table 1 Operator precedence in kP-Lingua guard expressions

Precedence level	Operator type	Operators	Associativity
1	Grouping	( )	N/A
2	Unary	- (negation), ! (logical NOT)	Right-to-left
3	Multiplicative	*, /	Left-to-right
4	Additive	+, -	Left-to-right
5	Relational	<, <=, >, >=, =, ! =	Left-to-right
6	Logical AND	&	Left-to-right
7	Logical OR	|	Left-to-right

above. The logical guards and communication links are automatically expanded to cover the range $1 \leq i \leq n$. This code demonstrates the use of rule indexation, guard indexation, multiset indexation, link indexation, and the definition of constants. The complete indexed specification is shown below:

```
#define k = 3, threshold = 10

type t1 {
  choice {
    <$threshold$c : b$i$ -> a$i$ (t2) . : 1 <= i <= k
    >=$threshold$c : b$i$ -> . : 1 <= i <= k
  }
}

type t2 {
  choice {
    a$i$ -> {b$i$, $i$c} (t1) . : 1 <= i <= k
    >=x$i$ : a$i$ -> {b$i$, $i$c} (t1) . : 1 <= i <= k
  }
}

c1 {@b$i$:1 <= i <= k@} (t1) .
c2$i$ {x$i$} (t2) . : 1 <= i <= k
c1 - c2$i$ . : 1 <= i <= k
```

The constants `k=3` and `threshold=10` determine, respectively, the number of indexed families and the control value for object `c`. Initially, compartment `c1` of type `t1` contains objects `b1, b2, b3`, while three compartments `c21, c22`, and `c23` of type `t2` each contain a distinct marker `x1`, `x2`, or `x3`. Links are established between `c1` and each of the `t2` compartments. The rules in type `t1` describe a threshold-dependent transformation. When the number of `c` objects is strictly less than 10, each object `b$i$` is rewritten into `a$i$` and sent to one of the connected `t2` compartments. Once the threshold is reached, `b$i$` is consumed without producing any further output. The rules in type `t2` return objects from `a$i$` to `b$i$`, while generating a number of `c` copies equal to the index `i`. Two families of rules exist: unconditional rules $\texttt{a\$i\$} \rightarrow \{\texttt{b\$i\$}, i \cdot c\}$ and rules guarded by the presence of marker `x$i$`, which apply only if the corresponding marker is present in the compartment. For `k=3`, the indexed specification expands into the following explicit model:

```
type t1 {
  choice {
    <10c : b1 -> a1 (t2) .
    <10c : b2 -> a2 (t2) .
    <10c : b3 -> a3 (t2) .
    >=10c : b1 -> .
    >=10c : b2 -> .
    >=10c : b3 -> .
  }
}

type t2 {
  choice {
    a1 -> {b1, 1c} (t1) .
    a2 -> {b2, 2c} (t1) .
    a3 -> {b3, 3c} (t1) .
    >=x1 : a1 -> {b1, 1c} (t1) .
    >=x2 : a2 -> {b2, 2c} (t1) .
    >=x3 : a3 -> {b3, 3c} (t1) .
  }
}

c1 {b1, b2, b3} (t1) .
c21 {x1} (t2) .
c22 {x2} (t2) .
c23 {x3} (t2) .
c1 - c21 .
c1 - c22 .
c1 - c23 .
```

3.2 Formal definition of expansion and semantic preservation

To formalize the indexation mechanism, we define an expansion function $\mathcal{E}$ that maps an indexed kP system specification $\Pi_{indexed}$ to a standard kernel P system Π_{flat}. The function $\mathcal{E}$ operates recursively on the syntactic constructs of kP-Lingua.

Let S be a set of indexed statements (rules, guards, or multisets) and I be the set of index variables with associated ranges R. The expansion function is defined such that:

$$\mathcal{E}(S, I) = \bigcup_{v \in \mathrm{dom}(R)} \mathcal{E}(S[v/i], I \backslash \{i\})$$

where $S[v/i]$ denotes the syntactic substitution of the index variable i with the concrete value v within the range defined by R. The base case occurs when $I = \emptyset$, at which point the statement is fully instantiated.

Preservation Claim: Since $\mathcal{E}$ is a purely syntactic transformation that unfolds parameterized patterns into explicit kP-Lingua structures without altering the underlying operational semantics, the expanded model preserves the behavior of the indexed model. Formally, for any indexed kP system model M and any temporal property ϕ, the indexed model satisfies the property if and only if the expanded model satisfies the expanded property:

$$M \models \phi \iff \mathcal{E}(M) \models \mathcal{E}(\phi)$$

This property is essential for the verification framework described in Sect. 4, ensuring that model checking results obtained on the expanded system $\mathcal{E}(M)$ remain valid for the high-level indexed specification.

4 Verification

Formal verification is an essential component in assessing whether computational models conform to their intended behavior. While recent efforts have successfully applied interactive theorem proving to variants, such as cP systems [27], within the kPWorkbench environment, verification of kernel P systems is supported through an automated property specification language based on temporal logic. Within the kPWorkbench environment, verification is supported through a property specification language based on temporal logic. The framework integrates both Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) model checkers – Spin [20] and NuSMV [6], allowing a broad class of behavioral properties to be expressed and automatically verified.

Verification in kPWorkbench is carried out on the corresponding mathematical models, which are explored exhaustively. The simulation generates execution traces capturing the evolution of multisets and compartment configurations, and these traces are subsequently checked against the specified temporal properties. Properties are written in kP-Query language, a natural-language syntax for properties, which are then automatically translated into LTL/CTL formulas. This abstraction enables researchers to specify reachability, safety, and liveness conditions in a concise and intuitive manner.

4.1 Indexation in verification

A distinctive new feature of verification in kPWorkbench is the integration of the unified indexation mechanism. This new indexation mechanism permits quantified, parameterised properties to be defined compactly, without requiring users to manually enumerate all instances. Indexed constants, symbols, and guards can appear directly in queries and are systematically expanded. This mechanism is particularly effective in models with homogeneous families of compartments or repeated structural motifs. For example, the following indexed invariant ensures that in every compartment of type `c2i`, the number of objects `xi` never exceeds 5:

```
@&, (c2$i$.x$i$ <= 5) : 1 <= i <= k @
```

After expansion, this yields `k` individual constraints, one for each compartment, each checked independently during verification. This approach aligns with earlier uses of parameterized properties in tissue-like and neural-like P systems, but substantially increases expressiveness and reduces specification overhead in kP systems. In the context of the indexed broadcasting model, the verification module was employed to confirm that message transmission completes successfully, that synchronisation among communicating compartments is preserved, and that control-related objects remain within their designated bounds throughout the computation. All properties were validated using the model checker integrated into kPWorkbench, demonstrating that the indexed model faithfully implements the intended communication protocol. This can be seen in Table 2.

The first property represents the persistent reachability of broadcast messages. This property expresses a core liveness requirement of the broadcasting system: the model must continuously reach states in which at least one broadcast object (`b1, b2, b3`) is available in compartment `c1`. In other words, the broadcasting activity must remain active throughout the execution, without stabilizing in a state where no broadcast object is ever produced again. By enforcing that such states occur infinitely often, the property guarantees that the communication protocol does not deadlock and that message dissemination is repeatedly re-triggered.

The second property constrains how resources accumulate in the system by linking the number of `c` objects in `c1` with the consumption of the acknowledgment objects `a1, a2`, and `a3`. Specifically, the system may contain fewer than 10 `c` objects at any time. However, if it exceeds that threshold, it is only permissible when all the `ai` objects have already been consumed. This specification captures a form of conditional safety, ensuring that the broadcast process does not progress toward completion (represented by growing `c` instances) unless all subordinate compartments have completed their required internal transitions.

The third property formalizes a causal dependency between different compartments of the system. Whenever the compartment `c21` produces an `a1` object, a local event signals a partial contribution to the broadcast–the system is required to eventually reflect its effect at the global level by producing at least one `b1` object in `c1`. The property ensures that local progress is not lost or blocked, and that every relevant intermediate computation ultimately contributes to the final broadcast. This captures the intended semantics of hierarchical communication within the system.

The fourth property ensures that certain global states can only be reached after the system has completed all necessary internal actions. If the central compartment `c1` accumulates 10 or more `c` objects, representing a substantial advancement in the broadcast protocol, then all acknowledgments `a1, a2`, and `a3` in the subordinate compartments must have already been consumed. This is a form of synchronization invariant: the system must not progress to a globally advanced state while some local sub-processes remain incomplete. It enforces a strict ordering between local termination and global progression.

The fifth property verifies a fundamental correctness condition: the counts of broadcast objects `b1, b2, b3` in `c1` must never become negative at any point during system execution. Negative multiplicities would be semantically invalid and would indicate an error either in rule application or in model encoding. By enforcing non-negativity as an invariant, the property acts as a structural sanity check, ensuring that the model respects the basic mathematical constraints of multiset-based computation.

The sixth property asserts that the broadcast process must eventually succeed in producing at least one broadcast object in `c1`. That is, the sum of all `bi` objects must become positive at some point. The property prevents executions in which the system remains indefinitely in an initial or idle state without ever producing a broadcast. It, therefore, confirms that the broadcasting mechanism is responsive and is capable of initiating message propagation under the given rules.

5 A solution to the subset sum problem

The Subset Sum problem is defined as a set of integers, where the goal is to determine whether there exists a non-empty subset whose elements sum up to a specific target value k. This problem is closely related to the Partition Problem; specifically, the Partition Problem can be reduced to the Subset Sum Problem by setting k equal to half the sum of all elements in the set. In this implementation, we define the Subset Sum problem using a set of types and rules, leveraging indexation to reference elements and their relationships systematically.

We provide a solution to the partition problem using a classical approach in membrane systems, also illustrated in [8]. The partition problem is formulated as follows: Let V be a finite set and *weight* be a function on V with positive integer values (this is an additive function). It is requested to find, if it exists, a partition of V, denoted V_1, V_2, such that $weight(V_1) = weight(V_2)$. A solution to this problem is provided in [8] by using a recognizer tissue P system with cell division and symport/antiport rules. We make the following notations: let $V = \{v_1, \ldots, v_n\}$, be a finite set, with $weight(v_i) = k_i$, where k_i is a positive integer, $1 \leq i \leq n$.

We build the following skP system, which depends on n, for solving the partition problem (i.e., checking whether there is a partition, V_1, V_2, with $weight(V_1) = weight(V_2)$) and working in the maximally parallel manner [14]:

$$\mathrm{skP}_P(n) = (A, L, IO, \mu, C_1, C_2, 0),$$

where

- A is the alphabet;
- $L = \{0, 1, 2\}$;
- IO consists of *yes*, *no*; at the end, after $n + 3$ steps, one of the two possible answers will be sent out;
- $C_1 = (1, w_{1,0}, R_1)$, $C_2 = (2, w_{2,0}, R_2)$, where $w_{1,0} = S$, $w_{2,0} = A_1\mathrm{code}(n)$, with code(n) = $v_1^{k_1} \ldots v_n^{k_n}$ being the code of the weights of the elements of V;
- μ is given by the graph with edge $(1, 2)$;
- R_1 and R_2 are given below:
- R_1 contains:

$$r_{1,1} : S \to (yes, 0)\{\geq T\}, \quad r_{1,2} : S \to (no, 0)\{\geq F < \mathtt{T}\};$$

  $r_{1,1}$ or $r_{1,2}$ sends into the environment the answer *yes* or *no*, respectively;
- R_2 contains membrane division rules:

  $$r_{2,i} : [A_i]_2 \to [B_i A_{i+1}]_2[A_{i+1}]_2, \quad 1 \leq i < n,$$

  $$r_{2,n} : [A_n]_2 \to [B_n X]_2[X]_2;$$

  these rules generate all the subsets of V in n steps (i.e., 2^n subsets), each of them being a potential V_1 (with V_2 as its complement).
  Rewriting rules:

  $$r_{2,i,j} : v_i v_j \to v\{= B_i \neq B_j = X \mid \neq B_i = B_j = X\},$$
  $$1 \leq i < j \leq n,$$

  $$r_{2,n+1} : X \to Y;$$

  Rewriting and communication rules:

  $$r_{2,n+2} : Y \to (F, 1)\{\geq v_1 \mid \ldots \mid \geq v_n\},$$

  $$r_{2,n+3} : Y \to (T, 1)\{< v_1 \ldots < v_n\}.$$

Table 2 List of properties for the Broadcasting model derived from the property language and their representations in different formats

Prop.	(i) Informal query, (ii) Indexed kP-Query, (iii) LTL, (iv) CTL
1	(i) The system repeatedly reaches states in which at least one of the objects b_1, b_2, or b_3 is present in c_1.
	(ii) `infinitely-often (@or, c1.b$i$ > 0 : 1<=i<=3@);`
	(iii) $G\big(F(((c1.b1 > 0 \vee c1.b2 > 0) \vee c1.b3 > 0) \wedge pInS) \vee \neg pInS\big)$
	(iv) `SPEC AG (EF (((c1.b1 > 0 | c1.b2 > 0) | c1.b3 > 0) & pInS) | !pInS)`
2	(i) Either c_1 has fewer than 10 c objects, or all the a_1, a_2, and a_3 objects in the related compartments have already been consumed.
	(ii) `always ((c1.c < 10) or (@and, c2$i$.a$i$ = 0 : 1<=i<=3@));`
	(iii) $G\big((c1.c < 10 \vee ((c21.a1 = 0 \wedge c22.a2 = 0) \wedge c23.a3 = 0)) \vee \neg pInS\big)$
	(iv) `SPEC AG ((c1.c < 10 | ((c21.a1 = 0 & c22.a2 = 0) & c23.a3 = 0)) | !pInS)`
3	(i) Whenever compartment c_{21} produces at least one a_1, this is eventually followed by the appearance of at least one b_1 in c_1.
	(ii) `(c21.a1 > 0) followed-by (c1.b1 > 0);`
	(iii) $G\big((c21.a1 > 0 \rightarrow F(c1.b1 > 0 \wedge pInS)) \vee \neg pInS\big)$
	(iv) `SPEC AG ((c21.a1 > 0 -> EF (c1.b1 > 0 & pInS)) | !pInS)`
4	(i) Whenever c_1 accumulates 10 or more c objects, all the a_1, a_2, and a_3 objects in compartments c_{21}, c_{22}, and c_{23} must have already been consumed.
	(ii) `always ((c1.c >= 10) implies (@and, c2$i$.a$i$ = 0 : 1<=i<=3@));`
	(iii) $G\big((c1.c \geq 10 \rightarrow ((c21.a1 = 0 \wedge c22.a2 = 0) \wedge c23.a3 = 0)) \vee \neg pInS\big)$
	(iv) `SPEC AG ((c1.c>= 10 -> ((c21.a1 = 0 & c22.a2 = 0) & c23.a3 = 0)) | !pInS)`
5	(i) At all times, the number of b_1, b_2, and b_3 objects in c_1 remains non-negative.
	(ii) `always (@and, c1.b$i$>= 0 : 1<=i<=3@);`
	(iii) $G\big(((c1.b1 \geq 0 \wedge c1.b2 \geq 0) \wedge c1.b3 \geq 0) \vee \neg pInS\big)$
	(iv) `SPEC AG (((c1.b1 >= 0 & c1.b2>= 0) & c1.b3>= 0) | !pInS)`
6	(i) Eventually, at least one of the b-type objects (b_1, b_2, or b_3) will appear in c_1.
	(ii) `eventually ((@+, c1.b$i$ : 1<=i<=3@) > 0);`
	(iii) $F\big(((c1.b1 + c1.b2 + c1.b3) > 0) \wedge pInS\big)$
	(iv) `SPEC EF (((c1.b1 + c1.b2) + c1.b3)> 0 & pInS)`

5.1 Implementation of the subset sum problem using indexation

This section presents the implementation of the Partition Problem using the new indexation features of kPWorkbench. Our goal is to illustrate how the platform's syntactic enhancements, indexed rule definitions, multisets, guards, file inclusion, and compartment links enable the creation of a compact, scalable, and modular kernel P system (kPS) model. In this implementation, several types of indexation mechanisms are employed. Rule indexation, where rules are generated automatically over a specified range of indices. This supports defining large families of similar rules, such as `v$i$, v$j$ -> v`, iterating over all index pairs `i` and `j`. Logical guard expressions are indexed to express conditional rule triggering based on the indexed object.

The kernel P system implementation for the Partition Problem is built using three main types of components. `T0` type is the output evaluator, collecting and interpreting the final result, while `T1` type is responsible for aggregating subset evaluations and determining whether any of them solve the partition problem. `T2` type needs to construct and evaluate specific subsets of the input multisets. The input multiset and initial configuration are imported via the CreatePartitionMultisets.kplt file, while component interactions are defined through indexed links (`t_0 - t_1 - t_2`). This design enables the simulation of all 2^n subsets through rule dynamics alone, without relying on membrane division or exhaustive manual enumeration. Following subset construction, the system generates $n+1$ instances of the control symbol `y`, encoding the target sum $k = 3$. These tokens serve as a reference for evaluating the total weight of the subset. Indexed choice rules then consume the `y` symbols based on the presence of `v_i` elements and an auxiliary marker `s1`. If the number of consumed `y` tokens exactly matches the total weight, a success token `t` is generated; otherwise, a failure token `f` is produced. The transition to state `s2` signals the conclusion of subset evaluation.

It is important to note that the evaluation performed inside each `T2` component checks only whether the selected subset attains the target value `k`, i.e., whether:

$$\sum_{v \in V_1} w(v) = k.$$

This corresponds to a subset-sum computation rather than a full partition check. In the classical formulation of the Partition Problem, one must ensure that two disjoint subsets have equal weight. Our model explicitly solves the Subset Sum Problem for a given target k. To use this model for the Partition Problem, a preprocessing step is required to calculate $k = \text{weight}(V)/2$. Consequently, the present construction solves the Partition Problem under the condition that the input target k is pre-calculated correctly.

Below is the kP-Lingua code implementing this structure using indexation:

```
#define n = 3, k = 3

type T0 {
}

type T1 {
    max {
        >=t : s -> yes (T0) .
        >=f & <t : s -> no (T0) .
    }
}

type T2 {
    max {
        a$i$ -> [b$i$, a$i+1$][a$i+1$] . : 1<=i<n
        a$n$ -> [b$n$, x][x] . : n<=n<=n
    }
    max {
        =b$i$ & !=b$j$ & =x | =b$j$ & !=b$i$ & =x : v$i$, v$j$ -> v .
        : 1<=i<=n, 1<=j<=n, i<j
    }
    max { x -> $n+1$y, s1 . } : n<=n<=n
    choice { <v$i$ & =s1 : y -> s . }  : 1<=i<=n
    choice { =$k$v  & =s1: y -> s . }  : n<=n<=n
    choice {
        s1 -> s2 .
        <y : $n+1$s, s2 -> t (T1) . : n<=n<=n
        >=y : y, s2 -> f (T1) .
    }
}

#include "CreatePartitionMultisets.kplt"
t0 - t1 - t2 .
```

The simulation is initialized by connecting the components through links between `t0`, `t1`, and the `t2` instances. It is worth noting the specific use of the expression `n<=n<=n` in the terminal rules of the above specification. This specific syntax cannot be omitted as it is currently required by the kPWorkbench indexing parser. Due to the implementation of the indexing mechanism, the parser mandates the presence of an explicit iterator constraint on the right side of the rule to correctly trigger the substitution of the variable `n` (declared globally via the `#define` statement) within the rule body on the left side. In the CreatePartitionMultisets.kplt file, we will have:

```
t0 {} (T0) .
t1 {s} (T1) .
t2 {a1, v1, 2v2, 3v3} (T2) .
```

In this context, a type defines the behavior of a family of compartments, analogous to membrane labels in classical P systems. Each instance declared in the initial configuration corresponds to a concrete compartment of that type. We utilize newly introduced indexation features in rule guards and multisets to obtain a concise, structured, and scalable solution. The model is written for $n = 3$ elements with a target subset sum $k = 3$. Given a multiset of weighted elements $V = \{v_1, v_2, v_3\}$ with weights $\{1, 2, 3\}$, the goal is to determine whether there exists a partition into two disjoint subsets

V_1 and V_2 such that:

$$\sum_{v \in V_1} w(v) = \sum_{v \in V_2} w(v) = \frac{w(V)}{2} = 3.$$

This is a canonical NP-complete decision problem. We encode it using the computational power of kernel P systems with the benefit of syntactic indexation. The model consists of three cooperating types. Type `T0` receives the final result (`yes` or `no`), type `T1` collects evaluation results and determines the correctness of candidate subsets, and type `T2` performs subset construction and verification.

Each `T2` object evaluates a different subset. Objects `b_i` select which elements are included. Indexation is used to declare rules that would otherwise require repetitive syntax concisely. We avoid redundant subset evaluations by allowing only unique pairs. We use indexed transitions to evaluate whether a given subset achieves the target sum `k`. In kP-Lingua, rules can be applied in maximally parallel or nondeterministic modes. The `max` keyword ensures that all eligible rules fire simultaneously within a compartment, enabling parallel subset construction and token propagation. The `choice` keyword, in contrast, applies only one eligible rule non-deterministically, allowing selective token consumption and unique evaluation of subsets. Together, these modes provide precise control over parallelism and branching in kernel P systems.

After all subset evaluations, the system collects outputs to determine the final truth value, and the decision is finalized in `T1`:

```
s1 -> s2 .
<y : $n+1$s, s2 -> t (T1) . : n<=i<=n
>=y : y, s2 -> f (T1) .
```

We instantiate eight `T2` components, each with a different subset of links. These objects evaluate all relevant non-redundant subsets.

This solution would traditionally require a large number of explicitly enumerated rules. The indexation mechanism allows developers to abstract over repetitive rule structures by specifying parameterized guards and actions.

In addition to the dynamic generation of subsets through indexed rewriting rules, we could consider the alternative case in which cell division is performed before the computation. In this approach, all compartments representing potential subsets are explicitly instantiated in the initial configuration.

```
t0 {} (T0) .
t1 {s} (T1) .
t2_1 {v1, 2v2, 3v3, @b$i$:1<=i<=3@, x} (T2) .
t2_2 {v1, 2v2, 3v3, @b$i$:1<=i<= 2@, x} (T2) .
t2_3 {v1, 2v2, 3v3, b1, b3, x} (T2) .
t2_4 {v1, 2v2, 3v3, b1, x} (T2) .
t2_5 {v1, 2v2, 3v3, @b$i$:2<=i<=3@, x} (T2) .
t2_6 {v1, 2v2, 3v3, b2, x} (T2) .
t2_7 {v1, 2v2, 3v3, b3, x} (T2) .
t2_8 {v1, 2v2, 3v3, x} (T2) .
```

Static generation externalizes the exponential complexity into the input size, making the system less flexible and harder to maintain. Dynamic generation internalizes the complexity within the simulation process but allows the model to remain scalable, modular, and reusable. In the static case, all 2^n possible subsets of the `n` elements must be explicitly defined as separate compartments at the outset. This results in an exponential growth of the initial configuration size and rule declarations. Specifically, for `n` elements, 2^n instances of type `T2` must be defined, each with a unique set of multiset contents corresponding to a subset. While simulation may proceed quickly, preparing such a static configuration becomes intractable as `n` increases, and the model loses generality and reusability.

5.2 Results

The main objective of this work is to assess how the unified indexation mechanism enhances the formal verification capabilities of kPWorkbench. More examples can be found in the kPWorkbench GitHub repository.[1] Table 2 summarizes the set of behavior properties considered for the indexed broadcasting model. For each property, we provide: (i) its informal description, (ii) its indexed kP-Query formulation, and (iii) the corresponding LTL and CTL representations automatically generated by kPWorkbench. These properties include liveness, safety, causality, and synchronisation constraints, and collectively illustrate how indexation supports the compact specification of families of structurally similar conditions.

The verification results confirm that all indexed queries were successfully translated into their expanded temporal logic forms and validated using the Spin (LTL) and NuSMV (CTL). This demonstrates that the indexation mechanism is correctly integrated throughout the verification guards, multisets, and temporal operators–and that it preserves semantic equivalence between compact indexed queries and their expanded forms. Importantly, properties involving symmetric or repeated structural patterns (e.g., Properties 1, 2 and 5) benefit significantly from indexation, reducing the specification size from several manually enumerated clauses to a single indexed expression.

While verification constitutes the core of the evaluation, we have also executed the indexed Partition Problem model in kPWorkbench as a supporting experiment. This model serves solely to confirm operational consistency and to illustrate that

[1] https://github.com/Kernel-P-Systems/kPWorkbench/tree/feature/kplingua-preprocessing/resources.

the indexed specification leads to the expected computational behaviour. In contrast, the formal verification outcomes summarised in Table 2 represent the principal evidence for the correctness and utility of the indexation mechanism.

6 Related works and comparison with other simulation frameworks

The development of membrane computing simulators has produced a variety of tools tailored to specific P system variants and use cases. The broader landscape and current software engineering challenges of these simulation frameworks, ranging from graphical interfaces to parallel simulation engines, have been extensively reviewed in recent literature within the *Journal of Membrane Computing* [40]. Many of these are reviewed in Zhang et al. [44], highlighting challenges, such as nondeterminism, parallelism, and scalability. The book presents detailed descriptions of several key simulation frameworks, such as P-Lingua, MeCoSim, Infobiotics Workbench, and JENA. These tools support the design, execution, and analysis of a wide range of membrane system variants, often incorporating advanced features such as rule prioritization, cell division, and stochastic behavior.

The Infobiotics Workbench (IBW) [3] supports both stochastic and deterministic simulation of biological systems using lattice-based and stochastic P systems. It includes modules for model checking, parameter optimization via evolutionary algorithms, and visualization of simulation data. Its focus is on multi-scale modeling and synthetic biology.

Another important simulator is the Java Environment for Nature-Inspired Approaches (JENA) [45]. It provides medium-level molecular dynamics simulations, modeling interactions, such as Brownian motion and chemical reactions, within membrane structures. It is especially suited for simulations that bridge biophysics and membrane computing, offering realistic representations of molecular behavior.

Snapse is a graphical simulator for spiking neural P systems, emphasizing user interaction and animation. It supports rule nondeterminism, simulation repeatability, and spike visualization [9]. Similarly, recent efforts published in the *Journal of Membrane Computing* have introduced web-based visual simulators for spiking neural P systems, further expanding the accessibility of these models through highly interactive interfaces [7].

In this section, we also present a comparative analysis of kPWorkbench with several established membrane computing simulation tools. The comparison highlights each tool's key features, with a particular focus on rule representation, indexation capabilities, multiset manipulation, file modularity, and support for formal verification.

UPSimulator is a versatile simulation environment for various P system models, including neural and tissue P systems [19]. It has a dedicated language, UPLanguage, that offers object-oriented abstractions for membranes, rules, and objects, and supports extensibility for novel computational models, similar to kPWorkbench. Each platform defines a domain-specific language to express rules and membrane configurations. Both tools provide notations for working with multisets of objects, a central feature of P system representations. UPSimulator supports multiset indexing using the '^' notation, while kPWorkbench uses the $ notation. They allow symbolic representation of multiple copies of the same object. One of the key distinguishing features of kPWorkbench is its comprehensive indexation mechanism, which supports the indexing of rules, guards, multisets, and even links between compartments.

MeCoSim is a simulator built on the P-Lingua framework, offering support for cell-like, tissue, and spiking neural P systems [41]. MeCoSim is inspired by earlier tools like the EcoSim product family [42]. MeCoSim and P-Lingua are closely integrated tools forming a cohesive simulation environment for membrane computing. While MeCoSim serves as a user-friendly graphical interface and execution engine, P-Lingua functions as the underlying domain-specific modeling language. Together, they support a variety of P system variants, including cell-like, tissue, and spiking neural P systems, with an emphasis on flexibility, extensibility, and readability. P-Lingua 5 allows users to define derivation modes, controlling how P systems evolve over time. This can be done using expressions like @model(variant), where derivation modes, such as maximal parallelism or custom modes, are defined. This customizability is crucial for users who need fine-grained control over the system behavior. It supports parametric and indexed rule specification, multiset manipulation, and the construction of derivation strategies [37]. This foundation enables MeCoSim and P-Lingua 5 to offer an interactive simulation environment, including support for rule prioritization, promoter-based evolution, and cell division mechanisms. Both tools adopt parametric rule formats for expressing indexed behaviors. This is an example of rules for solving the SAT problem using tissue P systems with promoters and cell division.

```
/* r1{i} */ [a{i}]'1 --> [t{i}]'1 [f{i}]'1 : 1<=i<=n;
```

In this rule, the index i ranges from 1 to `n`, meaning this rule applies individually to each `ai` where `i` is within the specified range. The index `i` uniquely identifies each instance of `a`, `t`, and `f`, ensuring that the system handles them distinctly for every value of `i`. If `i = 2`, the rewriting is `[a{2}]'1 -> [t{2}]'1 [f{2}]'1`. The MeCoSim and P-Lingua tools have the option to write the initial data using indexation.

Table 3 Feature Comparison between `UPSimulator`, `P-Lingua 5` and `kPWorkbench`

Feature	UPSimulator	P-Lingua 5	kPWorkbench
Rule definitions	Yes	Yes	Yes
Multiset objects	Yes	Yes	Yes
Compartments	Yes	Yes	Yes
Guards and conditions	No	No	Yes
Execution strategies	No	Yes	Yes
File inclusion	No	Yes	Yes
Link indexation	No	No	Yes
Formal verification	No	No	Yes
Variety of P system supported models	High	High	Medium

```
@ms (0) += a{i} : 1 <= i <= n+1;
```

Using the **+=** symbol, the specified objects will be added to the existing multiset in region 0. It ensures that any previously defined objects in `ms(0)` remain, and new objects are appended. The curly braces of `i` indicate parameterized objects, allowing multiple distinct instances. A difference between MeCoSim and kPWorkbench would be the indexation of the multisets. In kPWorkbench, users can add data to multisets during initial declaration without writing multiple lines of code. The indexation in kPWorkbench can be implemented using indexed rules that dynamically generate rule sequences, such as $[a_i \rightarrow b_{i+1}]$ for $1 \leq i \leq n$, thereby supporting large-scale, indexed simulations with minimal manual intervention. Both P-Lingua 5 and kPWorkbench support including code from external files, but they use different syntax for this functionality. In P-Lingua 5, the `@include` statement is used.

Both UPSimulator and P-Lingua 5 offer a broad coverage of P system variants, including cell-like, tissue, and spiking neural P systems, giving them an advantage in flexibility for simulating diverse computational models. kPWorkbench, while highly specialized in kernel P systems with advanced indexation and verification capabilities, focuses on a narrower class of P system variants. Furthermore, the theoretical mapping between various P system classes, such as spiking neural P systems, and kernel P systems, has been actively explored in recent JMC publications [13]. This reinforces the need for versatile, unified platforms like kPWorkbench that feature advanced syntactic mechanisms, such as indexation, to handle complex modeling paradigms effectively. A comparative overview is provided in Table 3.

7 Conclusions and future work

In this paper, we have introduced and illustrated several recent developments in the kPWorkbench platform, with a particular focus on the indexation mechanism for kernel P systems. The proposed enhancements enable the concise and scalable specification of rule-based systems through indexed definitions of rules, guards, multisets, and compartment links. This significantly improves model readability, reduces redundancy, and facilitates the construction and analysis of complex computational systems.

The practical utility of the indexation mechanism was demonstrated through the formalization and simulation of the Partition Problem, an NP-complete problem commonly used as a benchmark in computational modeling. The results confirm that indexed specifications not only streamline the model development process, but also maintain expressiveness and correctness within the kP system framework.

Furthermore, a comparative evaluation with other established simulation platforms highlights the distinctive strengths of kPWorkbench, notably its support for indexed guards, customizable execution strategies, and modular code inclusion via external files. The current indexation mechanism enables the specification of compact rules over contiguous integer ranges, offering an efficient approach for systems characterized by regular structural patterns. Nevertheless, its flexibility remains limited when addressing models that involve non-contiguous or irregular index sets, as commonly encountered in biological systems or distributed computing environments.

To enhance the kPWorkbench verification feature, we plan to introduce support for indexation within kPQueries, enabling compact and expressive property specifications in the temporal logic used for system verification.

The indexation mechanism in kPQueries is inspired by the indexation of rules and objects, offering a consistent and familiar syntax. It uses the special variable *i* and the delimited `@...@` notation, where a specified operation (e.g., `and`, `or`, `+`) is applied over a defined range of indices. Consider the following indexed query:

```
ltl: always @+, NC$i$.a : 0<i<=3@ = 1 ;
```

This statement expresses that the sum of the variable `a` across all compartments `NC1` to `NC3` must always be equal to 1. After applying the indexation, this is expanded as:

```
ltl: always (NC1.a + NC2.a) + NC3.a = 1 ;
```

To improve expressiveness, future versions will introduce *set-based indexation*, allowing users to explicitly specify index sets (e.g., `i in {0, 2, 4}`). This will simplify the modeling of sparse configurations and avoid redundant or conditional rule definitions.

We also plan to support *named vectors*, which enable reusable index sets declared once (e.g., `vector primes = {2, 3, 5, 7}`) and referenced throughout the model (e.g., `i in primes`). This enhances modularity and maintainability, especially in large-scale or domain-specific models where repeated index patterns occur.

These enhancements aim to make kPWorkbench more versatile and user-friendly. An example is already available in the public repository, demonstrating the broadcasting problem using indexed communication rules in a tree-structured membrane system. This shows the potential of the new features to model scalable, hierarchical signaling efficiently.

Author contributions F.I. and S.K. conceptualized the indexation mechanism and supervised the project. Ş.I. and I.M.N. implemented the software features in kPWorkbench. Ş.I. designed the Partition Problem and Broadcasting models and prepared the figures. Ş.I. wrote the main manuscript text, and all authors reviewed the manuscript.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Bakir, M. E., Ipate, F., Konur, S., Mierlă, L., & Niculescu, I. M. (2014). Extended simulation and verification platform for kernel P systems. In M. Gheorghe, et al. (Eds.), *15th international conference on membrane computing, LNCS 8961* (pp. 158–178). Springer.
2. Bakir, M. E., Konur, S., Gheorghe, M., Niculescu, I., & Ipate, F. (2014). High performance simulations of kernel P systems. In: *2014 IEEE 6th international symposium on cyberspace safety on cyberspace safety and security (HPCC/CSS/ICESS)* (pp. 409–412). IEEE.
3. Blakes, J., Twycross, J., Romero-Campero, F., & Krasnogor, N. (2011). The infobiotics workbench: An integrated in silico modelling platform for systems and synthetic biology. *Bioinformatics, 27*, 3323–3324. https://doi.org/10.1093/bioinformatics/btr571
4. Buiu, C., Vasile, C., & Arsene, O. (2012). Development of membrane controllers for mobile robots. *Information Sciences, 187*, 33–51. https://doi.org/10.1016/j.ins.2011.10.007
5. Cascado-Caballero, D., Diaz-del Rio, F., Cagigas-Muñiz, D., Rios-Navarro, A., Guisado, J. L., Pérez-Hurtado, I., & Riscos-Núñez, A. (2021). Marex: A general purpose hardware architecture for membrane computing. *Information Sciences*. https://doi.org/10.1016/j.ins.2021.10.064
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., & Tacchella, A. (2001). NuSMV version 2: An opensource tool for symbolic model checking. Technical Representative, ITC-IRST http://nusmv.fbk.eu/
7. Dupaya, A. G. S., Galano, A. C. A. P., Cabarle, F. G. C., & De La Cruz, R. T. (2022). A web-based visual simulator for spiking neural P systems. *Journal of Membrane Computing, 4*(1), 21–40.
8. Díaz-Pernil, D., Gutiérrez-Naranjo, M., Pérez-Jiménez, M., & Riscos-Núñez, A. (2010). A linear time solution to the partition problem in a cellular tissue-like model. *Journal of Computational and Theoretical Nanoscience, 7*, 884–889. https://doi.org/10.1166/jctn.2010.1435
9. Fernandez, A. D. C., Fresco, R. M., Cabarle, F. G. C., de la Cruz, R. T. A., Macababayao, I. C. H., Ballesteros, K. J., & Adorna, H. N. (2021). Snapse: A visual tool for spiking neural P systems. *Processes*. https://doi.org/10.3390/pr9010072
10. Frisco, P., Gheorghe, M., & Pérez-Jiménez, M. J. (Eds.) (2014) Applications of membrane computing in systems and synthetic biology. Springer.
11. Gheorghe, M., Ceterchi, R., Ipate, F., Konur, S., & Lefticaru, R. (2018). Kernel P systems: From modelling to verification and testing. *Theoretical Computer Science, 724*, 45–60. https://doi.org/10.1016/j.tcs.2017.12.010
12. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., & Pérez-Jiménez, M. J. (2013). Kernel P systems - Version I. In: 11th International Conference on Membrane Computing (pp. 97–124). Fénix Editora http://www.gcn.us.es/files/11bwmc/097_gheorghe_ipate.pdf
13. Gheorghe, M., Ipate, F., Kannan, K., Konur, S., Kuppusamy, L., Lefticaru, R., Mahendran, A., & Niculescu, M. I. (2025). Spiking neural P systems and kernel P systems. *Journal of Membrane Computing, 7*(4), 437–459.
14. Gheorghe, M., Ipate, F., & Konur, S. (2015). Solutions to the subset sum and partition problems using kernel P systems. *Annals of Bucharest University, Computer Science, 62*(2), 37–46.
15. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M. J., Ţurcanu, A., Valencia-Cabrera, L., García-Quismondo, M., & Mierlă, L. (2013). 3-col problem modelling using simple kernel P systems. *International Journal of Computer Mathematics, 90*, 816–830. https://doi.org/10.1080/00207160.2012.743712
16. Gheorghe, M., Konur, S., & Ipate, F. (2017). Kernel P systems and stochastic P systems for modelling and formal verification of genetic logic gates. In: A. Adamatzky (Ed.), Advances in unconventional computing, Volume 1: Theory (pp. 661–675). Springer. https://doi.org/10.1007/978-3-319-33924-5_25
17. Gheorghe, M., & Krasnogor, N. (2008). P systems applications to systems biology. *Bio Systems, 91*, 435–437. https://doi.org/10.1016/j.biosystems.2007.07.002
18. Gheorghe, M., Lefticaru, R., Konur, S., Niculescu, I. M., & Adorna, H. N. (2021). Spiking neural P systems: Matrix representation and formal verification. *Journal of Membrane Computing, 3*(2), 133–148. https://doi.org/10.1007/s41965-021-00075-1

19. Guo, P., Quan, C., & Ye, L. (2019). Upsimulator: A general P system simulator. *Knowledge-Based Systems, 170*, 20–25. https://doi.org/10.1016/j.knosys.2019.01.013
20. Holzmann, G. J. (1997). The model checker Spin. *IEEE Transactions on Software Engineering, 23*, 279–295.
21. Konur, S. (2014). Conventional verification for unconventional computing: A genetic XOR gate example. *Fundamenta Informaticae, 134*, 97–100. https://doi.org/10.3233/FI-2014-1093
22. Konur, S., & Gheorghe, M. (2015). A property-driven methodology for formal analysis of synthetic biology systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics, 12*(2), 360–371. https://doi.org/10.1109/TCBB.2014.2362531
23. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipate, F., & Krasnogor, N. (2014). Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. *ACS Synthetic Biology*. https://doi.org/10.1021/sb500134w
24. Konur, S., Mierlă, L., Ipate, F., & Gheorghe, M. (2020). kpworkbench: A software suit for membrane systems. *SoftwareX, 11*, 100407. https://doi.org/10.1016/j.softx.2020.100407
25. Lefticaru, R., Bakir, M. E., Konur, S., Stannett, M., & Ipate, F. (2018). Modelling and validating an engineering application in kernel P systems. In *19th international conference on membrane computing, LNCS 11270* (pp. 183–195). Springer.
26. Leporati, A., Manzoni, L., Mauri, G., Porreca, A., & Zandron, C. (2018). A survey on space complexity of P systems with active membranes. *International Journal of Advances in Engineering Sciences and Applied Mathematics*. https://doi.org/10.1007/s12572-018-0227-8
27. Mahendran, A., & Kuppusamy, L. (2021). Formal verification of cp systems using Coq. *Journal of Membrane Computing, 3*, 205–220.
28. Orellana-Martín, D., Valencia-Cabrera, L., Riscos-Núñez, A., & Pérez-Jiménez, M. J. (2019). A path to computational efficiency through membrane computing. *Theoretical Computer Science, 777*, 443–453. https://doi.org/10.1016/j.tcs.2018.12.024
29. Pang, S., Ding, T., Mao, X., & Xiong, N. N. (2021). Design and analysis of a decision intelligent system based on enzymatic numerical technology. *Information Sciences, 547*, 450–469. https://doi.org/10.1016/j.ins.2020.07.033
30. Păun, G. (2002). *Membrane computing—An introduction*. Springer.
31. Păun, G., & Romero-Campero, F. J. (2008). Membrane computing as a modeling framework. Cellular systems case studies. In M. Bernardo, P. Degano, & G. Zavattaro (Eds.), *Formal methods for computational systems biology* (pp. 168–214). Springer.
32. Păun, G., Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford Handbook of Membrane Computing*. Oxford University Press.
33. Peng, H., Li, B., Wang, J., Song, X., Wang, T., Valencia-Cabrera, L., Pérez-Hurtado, I., Riscos-Núñez, A., & Pérez-Jiménez, M. J. (2020). Spiking neural P systems with inhibitory rules. *Knowledge-Based Systems, 188*, Article 105064. https://doi.org/10.1016/j.knosys.2019.105064
34. Peng, H., Wang, J., Pérez-Jiménez, M. J., & Riscos-Núñez, A. (2019). Dynamic threshold neural P systems. *Knowledge-Based Systems, 163*, 875–884. https://doi.org/10.1016/j.knosys.2018.10.016
35. Păun, G. (2000). Computing with membranes. *Journal of Computer and System Sciences, 61*(1), 108–143. https://doi.org/10.1006/jcss.1999.1693
36. Pérez-Hurtado, I., Martínez-del Amor, M., Zhang, G., Neri, F., & Pérez-Jiménez, M. (2020). A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning. *Integrated Computer-Aided Engineering, 27*, 1–18. https://doi.org/10.3233/ICA-190616
37. Pérez-Hurtado, I., Orellana-Martín, D., del Amor, M. M., Valencia-Cabrera, L., & Riscos-Núñez, A. (2022). A new P-lingua toolkit for agile development in membrane computing. *Information Sciences, 587*, 1–22. https://doi.org/10.1016/j.ins.2021.12.003
38. Song, B., Li, K., Orellana-Martín, D., Pérez-Jiménez, M. J., & Pérez-Hurtado, I. (2021). A survey of nature-inspired computing: Membrane computing. *ACM Computing Surveys, 54*(1), 629–649. https://doi.org/10.1145/3431234
39. Song, B., Zhang, C., & Pan, L. (2016). Tissue-like P systems with evolutional symport/antiport rules. *Information Sciences*. https://doi.org/10.1016/j.ins.2016.10.046
40. Valencia-Cabrera, L., Pérez-Hurtado, I., & Martínez-del Amor, M. Á. (2020). Simulation challenges in membrane computing. *Journal of Membrane Computing, 2*, 392–402.
41. Valencia-Cabrera, L., & Song, B. (2020). Tissue P systems with promoter simulation with Mecosim and P-lingua framework. *Journal of Membrane Computing*. https://doi.org/10.1007/s41965-020-00037-z
42. Valencia-Cabrera, L., Wu, T., Zhang, Z., Pan, L., & Pérez-Jiménez, M. J. (2017). A simulation software tool for cell-like spiking neural P systems. *Romanian Journal of Information Science and Technology, 20*(1), 71–84.
43. Zhang, G., Pérez-Jiménez, M. J., & Gheorghe, M. (2017). *Real-life applications with membrane computing*. Springer.
44. Zhang, G., Pérez-Jiménez, M. J., Riscos-Núñez, A., Verlan, S., Konur, S., Hinze, T., & Gheorghe, M. (2021). *Membrane computing models: Implementations*. Springer.
45. Zhang, G., Pérez-Jiménez, M., Riscos-Núñez, A., Verlan, S., Konur, S., Hinze, T., & Gheorghe, M. (2021). Molecular physics and chemistry in membranes: The java environment for nature-inspired approaches (JENA). In *Membrane computing models: Implementations* (pp. 101–161). Springer. https://doi.org/10.1007/978-981-16-1566-5_5
46. Zhang, G., Shang, Z., Verlan, S., Martínez-del Amor, M., Yuan, C., Valencia-Cabrera, L., & Pérez-Jiménez, M. (2020). An overview of hardware implementation of membrane computing models. *ACM Computing Surveys, 53*, 1–38. https://doi.org/10.1145/3402456

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Ştefan Iacob is currently pursuing his Ph.D. at the University of Bucharest, where his research focuses on theoretical computer science and membrane computing. Prior to his doctoral studies, he earned his Master's degree from Transilvania University of Braşov. His academic interests encompass bio-inspired computing and P systems.



Florentin Ipate received his Ph.D. in Computer Science from the University of Sheffield, where his thesis on X-machines was nominated for the UK Distinguished Dissertation Award. He also holds an M.Sc. from Sheffield and an M.Eng. from the Politehnica University of Bucharest. An active member of the theoretical computing community, he currently serves as an editor for Springer's Journal of Membrane Computing. His academic leadership includes serving as the general chair for ICTAC in 2014, alongside ongoing programme committee service for major international conferences in software testing, formal methods, and bio-inspired computing, such as ICST, SEFM, and ICMC/CMC.



Ionuţ Mihai Niculescu is a Professor of Artificial Intelligence and Computational Modelling and Head of the School of Computing and Engineering at the University of Bradford. He also directs the Bradford–Renduchintala Centre for Space AI. His extensive research in Scientific AI develops machine learning-enhanced technologies and intelligent decision-support frameworks applied to engineering, manufacturing, and life sciences. Prof. Konur has a distinguished track record of securing over £6 million in research funding and delivering high-impact publications through interdisciplinary collaborations. He frequently serves on international grant panels, expert panels, and editorial boards. He is a Fellow of the Higher Education Academy and maintains active memberships in the International Membrane Computing Society, IEEE, and the British Computer Society.



Savas Konur is a computer scientist, researcher, and IT&C consultant. He holds a Ph.D., M.Sc., and B.Sc. in Computer Science from the University of Pitești. His doctoral thesis, Simulation of kernel P systems using communicating X-machines - Applications in FLAME, was supervised by Prof. Tudor Bălănescu and Prof. Florentin Ipate and was awarded The PhD Thesis of the Year 2018 by the International Membrane Computing Society (IMCS). Dr. Niculescu's primary research directions include computational models, P systems, parallel processing, and the simulation, verification, and testing of these models. His broader technical interests span parallel computing, quantum computing and information, the Internet of Things (IoT), and blockchain technologies.