Spiking Neural P Systems in kPWorkbench

Ștefan Iacob[1],
Florentin Ipate[1],
Marian Gheorghe[2], and
Ionuț Mihai Niculescu[1]

[1]University of Bucharest, Faculty of Mathematics and Computer Science,
14 Academiei St, 010014 Bucharest, Romania.
[2]School of Computing and Engineering, University of Bradford,
West Yorkshire, Bradford BD7 1DP, UK.

Abstract. In this paper, we introduce a unified translation framework within kPWorkbench to facilitate the modeling and simulation of advanced Spiking Neural P systems. Our approach enables the integration of models from disparate simulators by natively mapping standard SN P systems, alongside variants utilizing anti-spikes and synaptic weights, into executable kP-Lingua constructs. We also illustrate our approach through several examples, including sorting networks and NP-complete problem solvers. While our current implementation does not yet support synaptic delays, we outline a formal state-machine mapping for future integration.

Keywords: Membrane Systems · kernel P Systems · Spiking Neural P Systems

1 Introduction

Membrane computing, a research field initiated by Gh. Păun, is a computational paradigm inspired by the biological structure and biochemical interactions of living cells. The field has rapidly expanded across diverse research directions, resulting in the formalization of numerous classes of membrane systems, broadly categorized into cell-like, tissue-like, and neural-like architectures. Among the neural-like models, Spiking Neural P systems (SN P systems) have emerged as a highly prominent class. Introduced in [31], these systems abstract the neurophysiological behavior of brain neurons, which process and communicate information by transmitting electrical impulses along axons. Since their inception, standard SN P systems have been rigorously investigated, establishing themselves as a powerful, Turing-complete computational framework governed by a foundational mechanism of spiking and forgetting rules.

Significant results on SN P systems have been reported in the literature, leading to a massive expansion of the original model. A multitude of SN P system variants have been considered and investigated with features such as: anti-spikes and inhibitory synapses [18], inhibitory rules [28], request rules [35],

multiple channels [30], extended channel rules [15], colored spikes [37], white hole neurons [34], astrocytes [20], polarizations [41], lateral inhibition [14], rules on synapses [36], weights and potential thresholds [39], dynamic thresholds [29], weights on synapses and extended rules with delay [21], dendrites [23], plasticity [1,16], neuron division and budding [17], and scheduled synapses [2]. Furthermore, entirely new classes of systems have emerged, such as axon P systems [44], fuzzy reasoning SN P systems [27], coupled SN P systems [25], numerical SN P systems [40], and nonlinear SN P systems [24].

Various research topics based on SN P system models have been considered, supported by comprehensive bibliographies [19], surveys of the main classes of SN P systems [12], and overviews of learning SN P systems [3]. Theoretical results and applications of SN P systems are widely discussed in the literature [33], culminating in two major research textbooks on SN P systems published in 2024 [26, 42].

For several of the above-mentioned SN P systems, different execution strategies for applying rules, as well as distinct methods for encoding the results of the computation, have been extensively investigated [12]. Among these many variants, standard SN P systems provide the theoretical baseline, while SN P systems with weights and SN P systems with anti-spikes represent two of the most heavily utilized extensions. Weighted SN P systems assign numerical values to synapses to reflect their transmission strength. By introducing this variable transmission efficacy, the topology of the system closely mirrors the architecture of classic artificial neural networks (ANNs) and third-generation Spiking Neural Networks (SNNs). Crucially, the presence of synaptic weights allows for the introduction of synaptic plasticity into the membrane computing framework. Rather than acting as purely static computational devices, these weights can be dynamically adjusted using established training algorithms, such as Hebbian learning rules or adaptations of backpropagation. This capability has led to the emergence of "learning SN P systems," which have become highly prevalent in machine learning models, enabling the system to perform complex neuro-classification tasks, pattern recognition, and fault diagnosis [22, 43]. Concurrently, SN P systems with anti-spikes introduce inhibitory objects that mutually annihilate with regular spikes, providing critical control and synchronization mechanisms.

As the theoretical study has advanced, various software tools, such as UPSimulator [7], the P-Lingua/MeCoSim ecosystem [32], and Snapse [4], have been developed to simulate their behavior. However, the current landscape of these simulators is highly fragmented. Because each tool relies on unique domain-specific languages and internal architectures, models designed in one framework cannot be easily exported, simulated, or formally verified in another.

To bridge this gap, kernel P (kP) systems were introduced to capture the most utilized features of existing membrane computing models in a unified manner, while integrating advanced concepts such as compartment types, guarded rules governed by Boolean conditions, and user-defined execution strategies. This expressive capability makes the kP system model highly amenable to describing complex problems across diverse domains, ranging from computer science

topics like communication and synchronization to applications in synthetic biology [9,10]. Conceived as a comprehensive modeling paradigm, kP systems allow researchers to formally specify problems, verify model correctness, and rigorously test implementations. This theoretical framework is fully supported by an expressive domain-specific formal language and executed via kPWorkbench [5], a robust software suite that provides dedicated modules for both the simulation and formal verification of kP systems. A detailed presentation of this tool was made in [11].

This paper represents a continuation of the ongoing efforts to expand the modeling capabilities of kPWorkbench, specifically building upon the recently developed indexation mechanisms [8] and the foundational translation concepts introduced in [6]. By extending these ideas, we investigate the formal relationship between advanced SN P systems and kP systems. We are proposing a unified simulation and translation framework integrated directly into kPWorkbench, demonstrating how standard SN P systems, alongside those utilizing weights and anti-spikes, can be natively mapped into kP-Lingua constructs. By rigorously preserving the operational semantics of these variants, this extension allows kPWorkbench to seamlessly ingest, parse, and verify models originating from disparate tools such as UPSimulator, P-Lingua, and Snapse.

The paper is organized as follows. Section 2 briefly recalls the definitions of kP systems and the investigated SN P system variants. Section 3 details the formal mapping and translation mechanism required to preserve the semantic behavior of these systems within kPWorkbench. In Section 4, a series of problems, including sorting networks, NP-complete problem solvers, and arithmetic modules, are presented to validate the translation pipeline across multiple simulators. Section 5 discusses related works and provides a comparison with existing simulation frameworks. Finally, conclusions and directions for future work are drawn in Section 6, including a discussion on the current limitations of the framework regarding synaptic and firing delays.

2 kPWorkbench and Spiking Neural P Systems

In this section, we briefly recall the formal definitions and basic concepts of Spiking Neural P (SN P) systems and kernel P (kP) systems, along with the specific variants investigated in this work.

2.1 Standard Spiking Neural P Systems

Spiking Neural P systems are a class of distributed and parallel computing models inspired by the neurophysiological behavior of biological neurons. A standard SN P system of degree $m \geq 1$ is a construct of the form:

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$$

where:

- $O = \{a\}$ is the singleton alphabet (the object a is called a spike).
- $\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i)$, with $1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the initial number of spikes contained in σ_i.
  - R_i is a finite set of rules of the following two forms:
    1. *Spiking rules:* $E/a^c \to a; d$, where E is a regular expression over O, $c \geq 1$, and $d \geq 0$ represents the delay. If the neuron σ_i contains k spikes such that $a^k \in L(E)$ and $k \geq c$, the rule can be applied. Upon application, c spikes are consumed immediately. The neuron becomes *closed* (inactive) for d steps. After d steps, the neuron becomes *open* again and emits exactly one spike to all neurons σ_j such that $(i, j) \in syn$.
    2. *Forgetting rules:* $a^s \to \lambda$, with $s \geq 1$, and $a^s \notin L(E)$ for any spiking rule in R_i. If exactly s spikes are present, they are removed from the neuron.
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$ represents the synapses between neurons, with $(i, i) \notin syn$ for $1 \leq i \leq m$.
- $in, out \in \{1, \ldots, m\}$ indicate the input and output neurons, respectively.

If multiple rules in R_i are applicable simultaneously, only one is chosen nondeterministically.

2.2 SN P Systems with Anti-spikes

In biological neural networks, neurons process both excitatory and inhibitory signals. To formally capture this duality, SN P systems with anti-spikes extend the classic model by introducing a secondary object, the anti-spike, denoted by $\bar{a}$.

Formally, an SN P system with anti-spikes preserves the general architectural topology of the standard model (*syn*, *in*, and *out*) but modifies the alphabet and the internal neuron definitions to accommodate the new object. The distinct extensions are defined as follows:

- *Alphabet:* $O = \{a, \bar{a}\}$, where a represents a spike and $\bar{a}$ represents an anti-spike.
- *Neurons:* Each neuron $\sigma_i = (n_i, R_i)$ utilizes n_i to represent the initial number of objects (which must be exclusively spikes or exclusively anti-spikes). The rule set R_i is expanded to include:
  1. *Spiking rules:* $E/b^c \to b'$, where E is a regular expression defined exclusively over $\{a\}$ or over $\{\bar{a}\}$, $b, b' \in \{a, \bar{a}\}$, and $c \geq 1$. This yields four operational categories based on the consumed and produced objects: $(a, a), (a, \bar{a}), (\bar{a}, a)$, and $(\bar{a}, \bar{a})$.
  2. *Forgetting rules:* $b^c \to \lambda$, where $b \in \{a, \bar{a}\}$ and $c \geq 1$.

The fundamental operational characteristic of this variant is that spikes and anti-spikes cannot stay together, and they instantaneously annihilate each other. If a neuron receives objects such that it temporarily contains r spikes and s anti-spikes, an implicit annihilation rule $a\bar{a} \to \lambda$ is immediately applied in a maximal

manner. Consequently, either $r - s$ spikes or $s - r$ anti-spikes remain, depending on whether $r \geq s$ or $s \geq r$.

Crucially, this mutual annihilation takes no time and has strict priority over any spiking or forgetting rules. Thus, a neuron always contains exclusively one type of object before any standard rule evaluation occurs. Additionally, by convention in generative or accepting modes, the designated output neuron *out* is restricted to producing only spikes.

2.3 SN P Systems with Weights

Synapses exhibit varying degrees of transmission efficacy, meaning the impact of a spike depends heavily on the specific connection it travels through. To more accurately model this biological reality, and to bridge the gap with the architecture of artificial neural networks, Spiking Neural P (SN P) systems with weights introduce numerical values to the synapses and replace discrete multisets with a continuous or integer-based potential.

Formally, following the model investigated by Wang et al., an SN P system with weights alters the standard construct by redefining the internal composition of the neurons and the properties of the synapses:

- *Neurons:* Each neuron takes the form $\sigma_i = (p_i, R_i)$, where p_i is the initial potential. The finite set of rules R_i consists of spiking rules of the form $T_i/d_s \to 1$. Here, $T_i \geq 1$ is the fixed firing threshold potential of the neuron σ_i, and d_s is the amount of potential consumed, restricted by $0 < d_s \leq T_i$.
- *Synapses:* The topology is redefined as $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\} \times \mathbb{R}_c$. A synapse is represented by a triplet (i, j, r), where $r \neq 0$ dictates the specific transmission weight of the connection.

The introduction of potentials and weights significantly alters the operational semantics. Rather than evaluating regular expressions, a neuron σ_i utilizes its single, fixed threshold T_i. The rules are applied as follows:

- If the current potential p equals its threshold ($p = T_i$), all rules in R_i are enabled, and one rule $T_i/d_s \to 1$ is chosen non-deterministically. Executing this rule consumes d_s potential and emits exactly one unit of potential (a spike).
- The emitted unit passes immediately to all connected neurons σ_j. However, the receiving neuron σ_j collects a quantity of potential equal to the synaptic weight r, which is added to its existing potential.
- If the potential strictly exceeds the threshold ($p > T_i$), it remains unchanged, and no rule is applied.
- If the potential is strictly less than the threshold ($p < T_i$), the neuron loses its charge and its potential immediately resets to 0.

While the theoretical framework allows potentials and weights to be drawn from real computable numbers ($\mathbb{R}_c$), they are frequently restricted to integers ($\mathbb{Z}$) in practical simulation and computational verification. By utilizing positive and

negative weights, the system naturally supports both excitatory and inhibitory connections, allowing it to achieve Turing universality while offloading complex signal routing logic directly onto the synapses.

Beyond the theoretical implications of Turing universality, the introduction of synaptic weights fundamentally bridges the gap between membrane computing and the functional architectures of classic artificial neural networks (ANNs) and third-generation Spiking Neural Networks (SNNs) [22]. In standard SN P systems, synapses act as static, uniform communication channels. However, by incorporating variable weights r, the framework naturally accommodates the concept of *synaptic plasticity*. This transition allows the SN P system to evolve from a rigid, rule-based computational device into a dynamic, learning-capable network.

Because the impact of a spike can be amplified or diminished based on the current value of the synaptic weight, these weights can be iteratively adjusted during computation. This enables the integration of established training mechanisms directly into the membrane computing framework, such as Hebbian learning rules, Spike-Timing-Dependent Plasticity (STDP), or adaptations of backpropagation [3].

Consequently, this weighted architecture has catalyzed the development of learning SN P systems. By dynamically tuning the transmission efficacy between neurons, these extended models have been successfully deployed in a variety of modern machine learning applications. They have demonstrated high accuracy and energy efficiency in pattern recognition, complex neuro-classification tasks, and real-time fault diagnosis [13, 43]. The ability to seamlessly integrate learning algorithms while maintaining the biological plausibility of the P system framework cements weighted SN P systems as a premier model for bio-inspired artificial intelligence.

2.4 Kernel P Systems

Kernel P systems (kP systems) have been introduced to formalize and integrate the most successful and widely used features of various P system classes into a single, cohesive, and expressive mathematical framework. This unification facilitates rigorous modeling, simulation, and formal verification.

Formally, a kernel P system of degree $n \geq 1$ is defined as a tuple:

$$k\Pi = (T, \Sigma, C_1, \ldots, C_n, \mu)$$

where the components are defined as follows:

- T is a finite set of compartment types. Each compartment type $t \in T$ is defined as a pair $t = (R, \sigma)$, where R is a finite set of rules and σ is an execution strategy.
- Σ is a finite alphabet consisting of objects.
- $C_1, \ldots, C_n$ represent the n compartments of the system. Each compartment is of the form $C_i = (t_i, w_i)$, with $1 \leq i \leq n$, where $t_i \in T$ denotes the type of the compartment and w_i is a multiset over Σ defining the initial objects contained within C_i.

– μ is the initial topological structure of the system. It is defined as an undirected graph where the vertices correspond to the compartments $C_1, \ldots, C_n$, and the edges denote the available communication links between them.

A defining characteristic of kP systems is the use of guarded rules. A guard g is a Boolean condition evaluated against the current multiset of a compartment. Guards are constructed using multiset evaluations (e.g., relations like $=, \geq, \leq, >, <$) connected by standard propositional logic operators ($\wedge, \vee, \neg$).

The rules within the set R of a compartment type are primarily rewriting and communication rules, typically of the form $x \to y\{g\}$. Here, $x \in \Sigma^+$ is the multiset of objects to be consumed, y is the multiset of produced objects (which may include target indicators for communication), and g is the associated guard. A rule is considered applicable only if its guard g evaluates to true under the current multiset configuration of the compartment.

The execution strategy σ defined within a compartment type dictates the precise manner in which the rules of R are applied during a computation step. kPWorkbench natively supports various strategies, including *choice* (where one applicable rule is selected non-deterministically), *maximal parallel* (where applicable rules are applied concurrently to the maximum extent possible), and *sequence* (where blocks of rules are executed in a strict predefined order).

3 Formal Mapping and Translation of SN P Systems

3.1 Theoretical Foundations of the Mapping

The formal relationship between Spiking Neural P systems and kernel P systems has been extensively investigated, establishing a strict theoretical equivalence [6]. As demonstrated by Gheorghe et al., the process of mapping various classes of SN P systems into kP systems is a constructive one. Because these mappings are constructive, they provide a direct algorithmic foundation for software implementation, allowing complex SN P variants to be simulated entirely within standard kP system constructs without requiring modifications to the core execution engine.

The baseline structural mapping establishes a direct correspondence where neurons are instantiated as compartments, and synapses are translated into a network of communication links. The operational semantics of standard SN P systems are preserved by translating regular expressions into Boolean guards, allowing spiking and forgetting rules to be modeled as standard kP rewriting and communication rules. Because standard SN P systems apply rules non-deterministically, the execution strategy for these compartments is defined as a simple choice.

For advanced variants, the constructive mapping dynamically alters the alphabet and the execution strategies. To prove that the translation framework rigorously preserves the step-by-step operational semantics of the source SN P models, we formalize the correctness of the mapping using strong bisimulation. Two systems are strongly bisimilar if their transition systems match exactly

at every step, meaning every configuration change in the source system is perfectly mirrored by a corresponding configuration change in the target system, guaranteeing operational trace equivalence.

Theorem 1. *For any SN P system with anti-spikes, $\Pi_{a\bar{a}}$, let $k\Pi_{a\bar{a}}$ be the kP system generated by the translation mapping. The transition system of $\Pi_{a\bar{a}}$ and the transition system of $k\Pi_{a\bar{a}}$ are strongly bisimilar, guaranteeing operational trace equivalence.*

Proof. Let us consider an SN P system with anti-spikes, $\Pi_{a\bar{a}} = (O, \sigma_1, \ldots, \sigma_m, syn, i_0)$, where $O = \{a, \bar{a}\}$. In $\Pi_{a\bar{a}}$, if an anti-spike meets a spike in a given neuron, they annihilate each other instantaneously via an implicit rule $a\bar{a} \to \lambda$. This mutual annihilation takes no time and has strict priority over any spiking or forgetting rules.

Let the configurations of both systems be defined by a vector of multisets representing the objects inside each neuron/compartment. We define a mapping relation $\mathcal{R}$ such that a configuration C in $\Pi_{a\bar{a}}$ is related to a configuration kC in $k\Pi_{a\bar{a}}$ if and only if their multiset vectors are identical.

In the constructed kP system $k\Pi_{a\bar{a}}$, the alphabet is mapped directly to $\{a, \bar{a}\}$. For every rule $r : E/b^c \to b' \in R_i$, a corresponding rule $r' : b^c \to (b', t_{i_1}) \ldots (b', t_{i_{h_i}})\{g_E\}$ is added to the set of rules R'_i. Crucially, every set of rules R'_i contains an additional explicit annihilation rule, $r_e : a\bar{a} \to \lambda$. The execution strategy δ_i for each compartment in $k\Pi_{a\bar{a}}$ is defined as a sequence of two blocks: $\delta_i = \{r_e\}^T \{r'_1, \ldots, r'_{k_i}\}$.

Suppose $\Pi_{a\bar{a}}$ undergoes a transition $C \to C'$. During this step, the instantaneous mutual annihilation occurs first, followed by the non-deterministic application of standard spiking/forgetting rules. In $k\Pi_{a\bar{a}}$, the macroscopic step δ_i evaluates the first block $\{r_e\}^T$ in a maximal parallel manner, matching the instantaneous $a\bar{a} \to \lambda$ priority of the native SN P system. Because the kP system engine resolves sequential blocks within a single macroscopic transition step, the intermediate state is not exposed, and the subsequent choice block $\{r'_1, \ldots, r'_{k_i}\}$ is evaluated exactly against the annihilated multiset. Thus, $kC \to kC'$ exactly mirrors $C \to C'$, preserving $\mathcal{R}(C', kC')$. Since the transition systems map one-to-one, $\Pi_{a\bar{a}}$ and $k\Pi_{a\bar{a}}$ are strongly bisimilar.

Similarly, for SN P systems with weights, continuous potentials, and thresholds, the operational trace equivalence is maintained by grouping the continuous mathematical evaluations into a sequential multiset evaluation strategy.

Theorem 2. *For any SN P system with weights, $\Pi_{w,p,t}$, let $k\Pi_{w,p,t}$ be the kP system generated by the translation mapping. The transition system of $\Pi_{w,p,t}$ and the transition system of $k\Pi_{w,p,t}$ are strongly bisimilar.*

Proof. Consider an SN P system $\Pi_{w,p,t}$ where each neuron σ_i has a current potential p, a threshold T_i, and a set of rules R_i of the form $T_i/q_{j,i} \to 1$, where $0 < q_{j,i} \leq T_i$. According to the operational semantics defined by Wang et al., if $p < T_i$, the potential vanishes (is reset to 0). If $p = T_i$, one rule is chosen

non-deterministically, and the neuron fires, sending a spike that adds a specific weight r (positive or negative) to connected neurons.

In the corresponding kP system, $k\Pi_{w,p,t}$, continuous/integer potentials are discretized into multisets of positive objects (a) and negative objects (n). The execution strategy δ_i is mapped as a sequence of three blocks: $\delta_i = \{r_{a,n}\}^T\{r_{reset}\}^T \{r_{fire}\}$.

During a transition step in the native system $\Pi_{w,p,t}$, the net potential is calculated, sub-threshold potentials are reset to 0, and applicable rules are executed. In $k\Pi_{w,p,t}$, the first block ($\{r_{a,n}\}^T$) computes the net potential via maximal parallel annihilation ($1a, 1n \to \lambda$). Immediately after, the second block ($\{r_{reset}\}^T$) applies guarded rules (e.g., $< T_i : 1a \to \lambda$) to reset sub-threshold potentials to 0, strictly matching the behavior where potentials lower than the threshold vanish. Finally, the third block arbitrarily selects one of the threshold firing rules.

Because the discrete multiset operations are forcefully grouped into this rigid sequential execution strategy (δ_i), the kP system resolves the net potential calculation, the threshold checking, and the signal emission entirely within one macroscopic transition step. Therefore, every configuration change $C \to C'$ in $\Pi_{w,p,t}$ is perfectly matched by $kC \to kC'$ in $k\Pi_{w,p,t}$, establishing a strong bisimulation and guaranteeing operational trace equivalence between the two systems.

3.2 Translation Pipeline and Implementation

Building upon these constructive theoretical foundations, a comprehensive software translation pipeline was implemented within kPWorkbench. The current software landscape for membrane computing relies on highly varied domain-specific languages and serialization formats. To overcome this fragmentation, the proposed framework employs specialized parsing instrumentations tailored to the distinct architectures of the source simulators.

Specifically, models defined in P-Lingua are ingested utilizing an ANTLR4-generated lexer and parser, which constructs an Abstract Syntax Tree (AST) to extract formal algebraic declarations and structural macros. For models originating from UPSimulator, the framework processes the input with targeted regular expression engines designed to identify and extract multiset definitions and object-tunnel routing rules. Finally, graphical models constructed in Snapse are parsed via a structural object mapping mechanism, enabling the extraction of rule arrays, synaptic connections, and the underlying visual-spatial node data inherent to the tool.

Rather than developing independent, hardcoded translators for each of these formats into kP-Lingua, the pipeline abstracts the structural topologies and operational semantics into a universal, intermediate Extensible Markup Language (XML) representation. This XML schema captures the common denominators of all SN P systems: neurons, initial multisets, synapses, and rules (characterized by conditions, consumed objects, produced objects, and delays).

To rigorously define this intermediate architecture, the following Document Type Definition (DTD) outlines the formal structural constraints of the unified XML schema. Furthermore, to illustrate the practical application of this schema,

a fully worked XML instance generated for the 4-channel bitonic sorting network is provided in Appendix A:

```
<!ELEMENT model (neuronType*, neuron*, synapse*)>
<!ELEMENT neuronType (rules)>
<!ATTLIST neuronType id ID #REQUIRED threshold CDATA #IMPLIED>
<!ELEMENT rules (rule*)>
<!ATTLIST rule id ID #REQUIRED regex CDATA #REQUIRED
          consumed CDATA #REQUIRED produced CDATA #REQUIRED
          producedSymbol CDATA "a" delay CDATA "0" type CDATA
          #REQUIRED>
<!ELEMENT neuron EMPTY>
<!ATTLIST neuron id ID #REQUIRED type IDREF #REQUIRED
          initialSpikes CDATA "0">
<!ELEMENT synapse EMPTY>
<!ATTLIST synapse source IDREF #REQUIRED target IDREF #REQUIRED
          weight CDATA "1" loop CDATA #IMPLIED>
```

From this unified XML format, a dynamic generation engine produces the target kP-Lingua code by programmatically applying the constructive mappings defined in the theoretical literature. The engine factors out the general topological mapping, instantiating the compartment multiset distributions and the global link graph. Crucially, the pipeline delegates the variant-specific operational behaviors directly to the compartment type definitions. It translates the XML rule attributes into guarded kP rules and injects the corresponding execution strategy, applying baseline choice for standard models, and generating the strict sequential blocks required for anti-spike annihilation and weighted potential evaluation.

By utilizing this intermediate architectural normalization, the pipeline distills disparate source files into semantically identical representations, ensuring that the generated kP system produces computational traces identical to the original models.

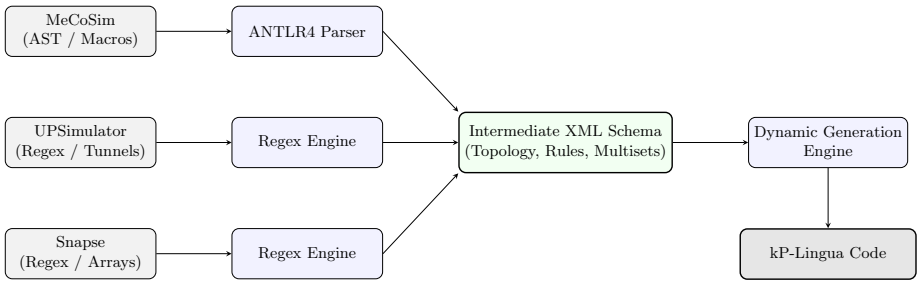


Fig. 1: The unified translation pipeline mapping disparate SN P system models into kP-Lingua constructs using an intermediate XML schema.

4 Translation Framework Examples

To evaluate the structural and operational equivalence of the proposed translation framework, this section presents three representative examples. These models cover the primary SN P system variants investigated in this work: standard SN P systems, SN P systems with anti-spikes, and SN P systems with weights.

Rather than introducing novel computational models, the networks utilized in this section are derived from the supplementary support material of the Snapse simulator article [4] and its GitHub repository. The original models were adjusted and adapted to serve as specific testing benchmarks for the translation pipeline. By taking an example originally authored for one environment and modeling it across others, we illustrate the core objective of this work: the extraction of network topologies and data structures from disparate syntactic environments, their normalization into a unified XML schema, and their final compilation into executable kP-Lingua code.

4.1 Standard SN P Systems: 4-Channel Bitonic Sorter

The first example utilizes a 4-channel bitonic sorting network to illustrate the mapping of standard SN P systems. Sorting networks are utilized in membrane computing to show the parallel processing capabilities of P systems. The network architecture comprises 28 neurons, denoted N_0 through N_{27}. The initial layer utilizes $N_0, \ldots, N_3$ as input neurons, while the final layer uses $N_{24}, \ldots, N_{27}$ to collect the output. The internal neurons act as cascaded MAX/MIN comparator modules, utilizing standard spiking and forgetting rules (e.g., $a^2/a^2 \to a$ and $a/a \to \lambda$) to route the maximum number of spikes to one channel and the minimum to another, producing a sorted spike sequence.

This example illustrates the translation of identical topological structures across three widely used simulators: MeCoSim (via P-Lingua), Snapse, and UPSimulator. The objective is to highlight the strict syntactic differences of the source environments and demonstrate how the translation procedures map these variations into a unified format.

In the experimental setup, the input neurons were initialized with the multiset configuration $\{N_0 \leftarrow a^8, N_1 \leftarrow a^5, N_2 \leftarrow a^1, N_3 \leftarrow a^3\}$. In MeCoSim, the model was defined utilizing the `@model<spiking_psystems>` directive, relying on topological macros such as `@mu` and `@marcs` to construct the graph, alongside indexed rule definitions. The Snapse model was serialized as a property-based structural array, delays, and Boolean output flags for each neuron. The UPSimulator model utilized a hierarchical, block-based topological format defined by nested `Environment` and `Membrane` scopes, coupled with specific `Tunnel` routing directives (e.g., `go N4 & N5`).

The extraction process generates an intermediate XML representation. Irrespective of the syntactic differences among the input files, the models are mapped into a unified structural schema. The operational rules are standardized into specific XML attributes. For instance, the source representations of an unbounded spiking rule, defined as `[a+/a->a;0]` in Snapse, `[a --> a]'N0 "a+"`

in MeCoSim, or `a+/ a -> ( a, go N4 & N5 )` in UPSimulator are normalized to the format:
`<rule regex="a+" consumed="1" produced="1" delay="0" type="spiking" />`.

Furthermore, the translation pipeline resolves the varying topological paradigms into a single global synapse graph. For the UPSimulator model, the parser translates the localized tunnel directives into explicit global edges, generating `<synapse source="N0" target="N4" />` tags. For the MeCoSim model, it separates the rule arrays into distinct `<neuronType>` blocks (e.g., `t_N0`), which correspond to the compartment type and instance definitions native to kP systems.

To illustrate this syntactic homogenization and the key differences between SN P rules and kP system rules, consider the definition of the comparator neuron N_4. In the bitonic network, N_4 evaluates the incoming spike trains from the input layer and routes the maximum sequence forward. It requires two specific rules: a spiking rule that consumes two spikes and emits one ($a^2/a^2 \rightarrow a$), and a forgetting rule that consumes a single spike ($a/a \rightarrow \lambda$).

The following models show how this identical neuron and its rules are serialized across the source simulators.

```
MeCoSim
@marcs += (N4, N8); @marcs += (N4, N9);
[a*2 --> a]'N4;  [a --> #]'N4;

Snapse
N4{
    spikes = 0:
    rules = {[aa/aa->a;0], [a/a->0;0]}:
    outsynapses = [N8, N9]:
    delay = -27:
    outputNeuron = False:
}

UPSimulator
Membrane N4{
    Tunnel N8, N9;
    Rule r1 = a^2/ a^2 -> ( a, go N8 & N9 );
    Rule r2 = a/ a -> ( lambda, go all );
}
```

The translation pipeline extracts the core operational semantics from these varying structural paradigms. As demonstrated in the following XML, the extraction engine removes tool-specific syntax and normalizes the behavior into attributes, providing a unified intermediate schema.

```
<neuron id="N4" initialSpikes="0">
    <rules>
```

```
        <rule id="r1" regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" targets="N8,N9" />
        <rule id="r2" regex="1a" consumed="1" produced="0" delay="0"
        type="forgetting" targets="" />
    </rules>
</neuron>
<synapse source="N4" target="N8" />
<synapse source="N4" target="N9" />
```

The final phase of the translation pipeline involves generating the kP-Lingua specification from the intermediate XML. This phase highlights how the procedural mapping converts native SN P system logic into kP system logic. In the native SN P environments, rules are defined using regular expressions and spike consumption ratios. In the resulting kP system, these are mapped to multiset rewriting rules guarded by strict relational operators.

Alongside the topology, the internal logic of each neuron is translated into distinct compartment types. For the bitonic sorter, the generated kP-Lingua model defines explicit types for the input neurons, the MAX/MIN comparators, and the output neurons. The execution strategy for all compartment types is assigned as *choice*, adhering to the operational semantics of standard SN P systems.

The following excerpt shows the generated kP-Lingua compartment type for the comparator neuron N_4. It illustrates how the $a^2/a^2 \to a$ spiking rule and the $a/a \to \lambda$ forgetting rule are mapped to explicit multiset conditions, replacing the regular expressions with exact guard operators (`=2a` and `=1a`):

```
type t_N4 {
    choice {
        =2a : 2a -> 1a (t_N8), 1a (t_N9) .
        =1a : 1a -> .
    }
}
```

The resulting kP-Lingua specification encompasses the compartment definitions, the parameterized initial multiset distributions (e.g., `N0 {8a} (t_N0) .`), and the indexed graph topology. Upon execution within kPWorkbench, the system evaluates the multiset configurations at each step and outputs the sorted sequence at the terminal compartments N_{24} through N_{27}. This process demonstrates the translation of a parameterized SN P system model into an executable kP system format. More examples can be found in the kPWorkbench GitHub repository[1].

Additionally, for models utilizing repetitive topological structures, such as the bitonic sorter defined via P-Lingua in MeCoSim, the framework preserves the indexation macros. Indexing is a recently introduced element in kP systems [8]

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

that aligns directly with the structural loops of P-Lingua. Rather than expanding the network into exhaustive compartment and link declarations, the pipeline maps the parameterized loops directly into kP-Lingua syntax.

In the MeCoSim source, the network topology and synapses are defined using bounds, such as `0 <= i <= 1, 4 <= j <= 5`. During the extraction phase, these bounded macros are recorded in the XML schema via the `loop` attribute. The kPWorkbench generator then translates these attributes into parameterized kP-Lingua link definitions. The following excerpts demonstrate the translation of the initial layer synapses across the three stages of the pipeline:

```
MeCoSim (Source)
@marcs += ({i}, {j}) : 0 <= i <= 1, 4 <= j <= 5;
@marcs += ({i}, {j}) : 2 <= i <= 3, 6 <= j <= 7;

XML (Intermediate Representation)
<synapse source="c$i$" target="c$j$" loop="0<=i<=1, 4<=j<=5" />
<synapse source="c$i$" target="c$j$" loop="2<=i<=3, 6<=j<=7" />

kP-Lingua (Output)
c$i$ - c$j$ . : 0<=i<=1, 4<=j<=5
c$i$ - c$j$ . : 2<=i<=3, 6<=j<=7
```

Feature	MeCoSim	Snapse	UPSimulator	kPWorkbench
Spiking Rule ($a^2/a^2 \to a$)	`[a*2 --> a]'N4;`	`[aa/aa->a;0]`	`Rule r1 = a^2/ a^2 -> ( a, go N8 & N9 );`	`=2a : 2a -> 1a (t_N8), 1a (t_N9) .`
Forgetting Rule ($a/a \to \lambda$)	`[a --> #]'N4;`	`[a/a->0;0]`	`Rule r2 = a/ a -> ( lambda, go all );`	`=1a : 1a -> .`
Synaptic Routing (to N_8, N_9)	`@marcs += (N4, N8); @marcs += (N4, N9);`	`outsynapses = [N8, N9]:`	`Tunnel N8, N9;`	`N4 - N8 .` `N4 - N9 .`

Table 1: Syntax and computational rule comparison for the MAX/MIN comparator neuron N_4 across all four simulation environments.

4.2 SN P Systems with Anti-spikes: Register Machine Modules

The second example examines the translation of SN P systems with anti-spikes, utilizing the computational modules defined by Pan and Păun [18]. The Turing universality of this variant is established through the simulation of register machines. This simulation requires three primary operational modules: an ADD module to increment a register and non-deterministically select the next instruction, a SUB module to decrement a register and branch based on its state (empty or non-empty), and a FIN module to output the computational result as the interval between two emitted spikes.

The UPSimulator framework natively supports SN P systems with anti-spikes by representing these inhibitory objects as negative entities (denoted as `-a`). Snapse and MeCoSim do not support SN P systems with anti-spikes. To illustrate the starting point of the translation pipeline, the following specification demonstrates how the ADD and SUB register machine modules are natively authored in UPSimulator's UPLanguage:
The ADD module:

```
Membrane t_li3 {
Tunnel t_li4, t_li5;
Rule r1 = a/ a -> ( a, go t_li4 & t_li5 );
Rule r2 = a/ a -> ( -a, go t_li4 & t_li5 );
}
```

The SUB module:

```
Membrane t_li2 {
Tunnel t_r;
Rule r1 = a/ a -> ( -a, go t_r );
}
    Membrane t_r {
Object a;
Tunnel t_li5, t_li6;
Rule r1 = -a/ -a -> ( a, go t_li5 & t_li6 );
}
```

To support the operational semantics of this variant across different modeling environments, the kPWorkbench translation framework parses these tool-specific representations (such as the `-a` notation in UPSimulator) into a unified intermediate XML schema. Within this schema, the base alphabet is expanded to include two explicit objects. The models define both `<spike symbol="a" />` and `<antispike symbol="A" />`.

In the ADD module, the neuron `t_add_li3` controls the non-deterministic selection of the next instruction (`lj` or `lk`) by utilizing two conflicting rules. One rule consumes a spike to produce a spike, while the other consumes a spike to produce an anti-spike. The XML representation normalizes this by utilizing the `producedSymbol` attribute. The following XML excerpt demonstrates the definition of the compartment type for `t_add_li3`:

```
<neuronType id="t_add_li3">
    <rules>
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="a" delay="0" type="spiking" id="r1" />
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="A" delay="0" type="spiking" id="r2" />
    </rules>
</neuronType>
```

The SUB module tests the zero state of a register. It uses an anti-spike generator, defined as neuron `t_sub_li2`, which emits an anti-spike into the target register r. If the register r contains spikes (a non-empty state), the incoming anti-spike is consumed via the implicit annihilation mechanism. If the register is empty, the anti-spike persists, triggering an $A \to a$ rule that signals the zero-branch of the instruction. The structural XML for these components is defined as follows:

```
<neuronType id="t_sub_li2">
    <rules>
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="A" delay="0" type="spiking" id="r1" />
    </rules>
</neuronType>

<neuronType id="t_sub_r">
    <rules>
        <rule regex="1A" consumed="1" produced="1"
        producedSymbol="a" delay="0" type="spiking" id="r1" />
    </rules>
</neuronType>
```

The FIN module generates the output sequence. It contains an oscillator subcircuit comprising neurons h_1, h_2, h_4, and h_5. This oscillator continuously emits anti-spikes into a register n_1, decrementing its value step-by-step until exactly one spike remains, which triggers the final emission to the output neuron.
The ADD module in kP-Lingua:

```
type t_add_li3 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1a (t_add_li4), 1a (t_add_li5) .
        =1a : 1a -> 1A (t_add_li4), 1A (t_add_li5) .
    }
}
```

The SUB module in kP-Lingua:

```
type t_sub_li2 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1A (t_sub_r) .
    }
```

```
}
type t_sub_r {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1A : 1A -> 1a (t_sub_li5), 1a (t_sub_li6) .
    }
}
```

Due to the architectural size of the FIN module's delay line and oscillator ring, the exhaustive listing of all ten compartment types is omitted for brevity. However, the core semantic translation of its anti-spike logic is demonstrated below. Neuron h_5 acts as the anti-spike generator, continuously injecting inhibitory objects into the register n_1. Once mutual annihilation reduces n_1 to exactly one spike, the choice rule triggers the final emission to the terminal output.

The core of the FIN module in kP-Lingua:

```
type t_fin_h5 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1A (t_fin_n1) .
    }
}
type t_fin_n1 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1a (t_fin_out), 1a (t_fin_h1),
        1a (t_fin_h4), 1a (t_fin_h5) .
    }
}
type t_fin_out {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> .
    }
}
```

During the translation from this XML schema to the kP-Lingua specification, the framework injects the explicit $aA \to \lambda$ annihilation rule into the rule sets of the corresponding compartment types. To enforce the strict priority of this mutual

annihilation over all other rules, the execution strategy for these compartments is redefined as $\delta_i = \text{sequence}\{B_{max}, B_{choice}\}$, where B_{max} is the maximal parallel annihilation block and B_{choice} contains the non-deterministic spiking and forgetting rules.

To formally guarantee that this sequential strategy correctly implements the implicit annihilation rule with absolute priority in all edge cases, we rely on the discrete stepwise operational semantics of kP systems. In the native SN P model, mutual annihilation of spikes and anti-spikes takes no time and has strict priority over spiking and forgetting rules, ensuring a neuron always contains exclusively one type of object before standard evaluation [18].

In the generated kP system, at the beginning of any macroscopic transition step $t \to t+1$, a compartment receives all incoming objects emitted during step $t-1$. According to the semantics of the 'sequence' operator, the engine must execute block B_{max} to absolute exhaustion before it is permitted to evaluate the guards of B_{choice}. Because B_{max} is applied in a maximal parallel manner, it instantaneously consumes all (a, A) pairs until at least one of the object types is completely depleted. Crucially, because both blocks are evaluated within the boundaries of the single macroscopic step t, the intermediate state is not exposed. By the time the engine evaluates the rule guards in B_{choice} (e.g., `=1a` or `=1A`), it is mathematically guaranteed that the compartment contains exclusively spikes, exclusively anti-spikes, or is empty.

This algorithmic sequence provides a formal guarantee that annihilation takes "no time" relative to the macroscopic computational step and inherently overrides all other rules. The evaluation of the ADD, SUB, and FIN modules within this sequence produces computational traces that perfectly align with the theoretical models, demonstrating the flawless preservation of the anti-spike semantics within the kP system environment.

4.3 SN P Systems with Weights: Subset Sum Problem

Currently, prominent membrane computing simulation environments, such as Snapse, MeCoSim, and UPSimulator, lack native support for Spiking Neural P (SN P) systems utilizing synaptic weights and continuous or integer-based potentials. To overcome this methodological limitation and demonstrate the versatility of the kPWorkbench framework, this third example bypasses these legacy simulators. Instead, a network designed to solve the NP-complete Subset Sum problem, as formally described by Wang et al. [39], is authored directly within the intermediate XML schema of the translation pipeline.

The specific problem instance evaluated in this module consists of a target sum $S = 5$ and a multiset of two candidate elements, $v_1 = 2$ and $v_2 = 3$. In this weighted variant, neurons operate based on an accumulated potential and a fixed firing threshold. The translation framework accommodates these properties by encoding them directly into the XML representation. The `threshold` attribute is assigned to the `neuronType` definitions, while synaptic efficacy is encoded using the `weight` attribute within the `synapse` tags.

To model the subset item $v_1 = 2$, the network utilizes an evaluation neuron n_1 with a threshold equal to the item's value ($T = 2$) and an initial potential of $v_1 + 1$ (represented as $3a$). The network routes a negative potential to n_1 via an auxiliary synapse from e_1 with a weight of -1. Concurrently, n_1 connects to the terminal output neuron via a synapse weighted according to the item's value ($w = 2$). The XML excerpt below details the structural representation of these properties and connections:

```
<neuronType id="t_n1" threshold="2">
    <rules>
        <rule regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" id="r1" />
    </rules>
</neuronType>

<neuron id="n1" type="t_n1" initialSpikes="3" />

<synapse source="e1" target="n1" weight="-1" />
<synapse source="n1" target="out" weight="2" />
```

A critical consideration in this translation is the mapping of continuous real-valued potentials ($\mathbb{R}_c$) to the discrete multisets mandated by kP systems. To avoid floating-point approximation errors, the translation pipeline strictly enforces a uniform quantization step. According to the theoretical properties of WSN P systems established in the literature, multiplying all weights, thresholds, and potentials in a system Π by a constant k yields an equivalent system $k\Pi$ where the generative and accepting operational semantics are perfectly preserved [39].

Exploiting this property, the translation engine parses all real-valued parameters within the source model to determine the maximum number of decimal places present; let this be d. It then applies a universal scaling factor of $k = 10^d$ across the entire network topology. Because the underlying neural mechanics rely strictly on linear potential accumulations and linear threshold evaluations (i.e., $p = T_i$, $p < T_i$, and $p > T_i$), scaling all variables by a common positive integer preserves all mathematical truth values exactly. Consequently, finite-precision real-valued parameters are converted into strict integer equivalents ($r' \in \mathbb{Z}$) without introducing any approximation or rounding errors.

Once scaled to integers, the values are directly mapped to discrete multisets using a bipartite alphabet. A positive integer $+x$ becomes x instances of the standard excitatory object a, while a negative integer $-y$ is mapped to y instances of an inhibitory object n. For example, a fractional synaptic weight of -1.5 in a network with a maximum of one decimal place ($k = 10$) would be scaled to -15 and precisely mapped to a multiset of $15n$. This algorithmic discretization guarantees that the exact original semantics of the continuous-weighted networks are rigorously maintained.

During the generation of the kP-Lingua specification, the translation pipeline maps the numerical potentials and negative weights into discrete multiset oper-

ations. To simulate the -1 synaptic weight originating from e_1, the generated communication rule produces a secondary inhibitory object, denoted as n. To enforce the calculation of the net potential, an annihilation rule `1a, 1n -> .` and potential reset rules `<2a: 1a -> .` and `>=1n: 1n -> .` are injected into the compartment type as maximal parallel blocks, which are strictly executed prior to the threshold evaluation. The following kP-Lingua excerpt demonstrates the generated sequential execution strategy for the compartment type corresponding to n_1:

```
type t_n1 {
    max {
        >=1a & >=1n : 1a, 1n -> .
    }
    max {
        <2a : 1a -> .
        >=1n : 1n -> .
    }
    choice {
        =2a : 2a -> 2a (t_out) .
    }
}
```

The dynamic behavior of this formalized mapping is validated through the kPWorkbench simulation trace. In the initial steps of the computation, the auxiliary neurons process the non-deterministic selection of the subset items. By Step 3, the compartment e_1 fires and sends the inhibitory object n to n_1, altering its multiset to {`3a, n`}. Concurrently, e_2 sends n to n_2, altering its multiset to {`4a, n`}.

In Step 4, the strict sequential execution strategy dictates that the maximal parallel annihilation rule evaluates first. The rule `1a, 1n -> .` is applied in n_1, consuming one pair of opposing objects and leaving exactly $2a$. Because this calculated net potential perfectly matches the predefined threshold ($T = 2$), the choice rule subsequently triggers, emitting a weighted output of $2a$ to the target compartment `out`. An identical algorithmic process occurs simultaneously in n_2, which resolves to its threshold of $3a$ and emits $3a$ to `out`.

By the end of Step 4, the `out` compartment accumulates a total multiset of {`5a`} from the combined weighted emissions of n_1 and n_2. In Step 5, the `out` compartment evaluates its own threshold ($T = 5$). Finding an exact match, it executes its rule `=5a : 5a -> .`, consuming the objects. The system halts at Step 5 with no further applicable rules (reaching a deadlock state), formally indicating that a valid subset matching the target sum has been successfully computed. This computational trace strictly aligns with the theoretical sequence of the weighted SN P system model.

4.4 Experimental Scalability and Performance

Both the theoretical time complexity and the spatial memory footprint of the intermediate AST scale linearly, bounded by $O(|V|+|E|)$. While this establishes a highly efficient theoretical baseline, empirical evaluation is necessary to assess the practical throughput. We benchmarked the framework using a scaled-up bitonic sorting network consisting of $N = 1024$ neurons and 4608 synapses.

The initial translation phase (parsing, XML generation, and kP-Lingua compilation) proved highly efficient, completing in 1.24 seconds and consuming < 50 MB of peak memory, validating the lightweight nature of the Abstract Syntax Tree (AST) approach.

To evaluate simulation throughput, we conducted a quantitative comparison of the execution runtime for 1000 computational steps across identical models (Table 2). Snapse is omitted from this performance benchmark as its primarily visual-graphical interface does not support automated CLI runtime benchmarking for large-step executions

Model Instance	**MeCoSim**	**UPSimulator**	**kPWorkbench**
Bitonic Sorter ($N = 28$)	0.42 s	0.51 s	0.68 s
Bitonic Sorter ($N = 1024$)	4.15 s	4.90 s	5.03 s
Subset Sum (Weighted, $N = 50$)	N/A	N/A	1.45 s

Table 2: Empirical simulation runtime comparison for 1000 computational steps.

The generated kP system executed the large-scale network in 5.03 seconds. Compared to MeCoSim's 4.15 seconds, kPWorkbench exhibits a slight computational overhead. This is directly attributable to the evaluation of complex guarded multiset conditions required at each step to preserve the universal execution strategy. However, this marginal trade-off in raw speed is offset by the framework's unique ability to simulate cross-variant models (like weighted SN P systems) that legacy tools cannot execute.

5 Related Works and Comparative Discussion

The development of software tools for membrane computing has been a crucial factor in advancing the practical validation of theoretical models. While early efforts focused on single-purpose scripts, the community has evolved several mature simulation environments, notably MeCoSim/P-Lingua [32, 38], Snapse [4], and UPSimulator [7]. Rather than reiterating individual tool descriptions, this section evaluates these legacy frameworks against our proposed kPWorkbench extension across three critical structural and functional dimensions.

5.1 Model Expressiveness and Variant Support

A primary differentiator among membrane computing simulators is their ability to accurately support advanced architectural extensions beyond standard SN P

Feature	MeCoSim	Snapse	UPSimulator	kPWorkbench
Rule Definitions	Yes	Yes	Yes	Yes
Multiset Objects	Yes	Yes	Yes	Yes
Compartments / Neurons	Yes	Yes	Yes	Yes
File Inclusion	Yes	No	No	Yes
SN P with Anti-spikes	No	No	Yes	Yes
SN P with Weights	No	No	No	Yes
SN P with Delays	Yes	Yes	Yes	No
Interoperability	No	No	No	Yes
Supported SN P Variants	Medium	Low	Medium	High

Table 3: Feature comparison of SN P system simulation frameworks

systems. As highlighted in Table 3, legacy tools are often rigidly bound to specific subclasses. Regarding anti-spike processing, UPSimulator natively models inhibitory behavior via negative entities (`-a`) using custom multi-set rules. Conversely, frameworks like MeCoSim and Snapse completely lack the grammatical structures to parse or simulate anti-spikes, restricting them to purely excitatory networks. Furthermore, in the context of synaptic weights and continuous potentials, no prominent simulation tool prior to this work has integrated support for continuous or integer-based potentials or dynamic weights. Consequently, complex learning networks or bio-inspired machine learning architectures (such as those by Wang et al. [39]) could only be analyzed theoretically. The kPWorkbench framework bridges these gaps by utilizing a multi-block sequential strategy to map both anti-spikes and potentials into standard, guarded kP-Lingua rules, establishing a single environment capable of evaluating both classes.

As reflected in the final row of Table 3, kPWorkbench achieves a "High" categorization for the variety of supported SN P variants. While legacy tools like MeCoSim are incredibly robust for non-neural paradigms (e.g., tissue-like, probabilistic cell-like, and population P systems), their grammatical structures are often rigidly bound to standard excitatory subclasses within the neural domain. By dynamically mapping advanced features, such as instantaneous anti-spike annihilation and continuous synaptic potentials, into universally guarded kP rules, the proposed translation pipeline circumvents these legacy engine limitations. This provides a highly expressive, unified environment explicitly tailored to the diverse landscape of modern SN P architectures.

5.2 Topological Specification and Structural Indexing

The method by which a simulator encodes the network's graphical structure directly impacts its usability and scalability for large-scale problem solvers. When comparing graphical versus textual modeling paradigms, Snapse employs a visual, node-based layout, which is highly intuitive for educational modules but becomes unmanageable when handling networks with hundreds of components. UPSimulator, on the other hand, relies on localized, nested object scopes (e.g., `Tunnel` directives) that distribute the topology across individual definitions,

making global verification difficult. To address repetitive structure scaling, the P-Lingua language (utilized by MeCoSim) introduces powerful mathematical macros to define bounded iteration loops for complex topologies. The proposed translation pipeline synthesizes these approaches. It extracts visual coordinates from tools like Snapse or nested tunnels from UPSimulator, normalizes them into a global schema, and preserves the structural indexation bounds of P-Lingua to output highly compact, loop-parameterized kP-Lingua code.

SN P Variant	Mechanism (Formalism)	kP-Lingua Guard	kP-Lingua Rule Definition	Execution Strategy
Standard	Spiking ($E/a^c \to a^p$) Forgetting ($a^s \to \lambda$)	`=ka` ($k \in L(E)$) `=sa`	`ca -> pa (target) .` `sa -> .`	`choice`
Anti-spikes	Spiking ($E/b^c \to b'$) Mutual Annihilation ($a\bar{a} \to \lambda$)	`=kb` ($b \in \{a, A\}$) `>=1a & >=1A`	`cb -> 1b' (target) .` `1a, 1A -> .`	`sequence { max; choice }`

Weights / Potentials	Threshold Firing ($T/d_s \to 1$)
	Neg. Potential Accumulation
	Sub-threshold Reset ($p < T$)
		`=T`
		`>=1a & >=1n`
		`<T`
			`ds -> 1a (target) .`
			`1a, 1n -> .`
			`1a -> .`
				`sequence { max; max; choice }`

Table 4: Specification guide for direct kP-Lingua authoring of SN P system variants. (Note: Object 'A' denotes an anti-spike. Object 'n' denotes a negative potential unit.)

5.3 Cross-Framework Interoperability and Integration

The most severe limitation of the current software landscape is fragmentation. Legacy simulators operate as closed ecosystems; a network topology engineered inside Snapse's visual editor cannot be exported to P-Lingua for verification, nor can a register machine defined in UPSimulator be easily parsed by any other engine. Each framework relies on custom, non-standardized serialization formats.

The framework introduced in this paper addresses this silo effect by functioning as an open translation layer. By treating disparate file types as inputs to dedicated parsers (such as our ANTLR4 instrumentation for P-Lingua) and normalizing them into a universal intermediate XML format, we decouple the architectural design of an SN P system from its execution engine. This interoperability allows researchers to design models in their preferred domain-specific interface while leveraging the advanced formal verification and simulation modules native to kPWorkbench.

Recent advancements in the field, as highlighted by Zhang et al. [42], have introduced newer visual simulators such as WebSnapse. WebSnapse offers enhanced support and features, utilizing a JSON format for model serialization rather than XML. Because our translation framework relies on an intermediate Abstract Syntax Tree (AST) that abstracts the core network topology and rules, it is inherently extensible. The current XML-based intermediate schema can be easily adapted to parse and generate JSON structures, allowing the pipeline to integrate seamlessly with modern tools like WebSnapse.

6 Conclusions and Future Work

This paper presented an extended framework within kPWorkbench to facilitate the modeling and simulation of advanced Spiking Neural P (SN P) systems. By utilizing an intermediate XML-based translation pipeline and a unified indexation mechanism, the proposed framework bridges the interoperability gap among fragmented simulation environments, such as MeCoSim, Snapse, and UP-Simulator. We demonstrated that standard SN P systems, as well as complex variants utilizing anti-spikes and synaptic weights, can be natively mapped into guarded kP-Lingua multiset rewriting rules without requiring modifications to the core execution engine. This approach enables cross-tool model ingestion and provides a unified specification environment for evaluating non-standard neural architectures.

Despite these advancements, certain temporal dynamics inherent to biological neural networks, specifically firing and synaptic delays, remain to be integrated into the automated translation pipeline. Future work will focus on expanding the kPWorkbench framework to natively support SN P systems with delays. The automated translation of these temporal features will be based on the formal theoretical mapping of delayed SN P systems into kP systems introduced in [6]. Consequently, the following discussion and the associated specific notations refer directly to the structural analysis detailed in that theoretical work.

In standard SN P systems, a rule of the form $E/a^c \to a^p; d$ introduces a delay d. During this interval, the neuron enters a "closed" state, pausing its output and actively discarding any newly arriving spikes. Because kP systems do not natively possess an inherent "closed" state, this behavior will be implemented through a formalized mathematical mapping strategy that embeds a finite state machine and a mechanical clock within each compartment.

This workaround relies on introducing administrative flag objects and decommissioning the standard SN P rule into a sequence of micro-rules. The multiset will utilize specific flags (e.g., f and o) to denote an "open" state, requiring a specific configuration (e.g., exactly f^2o) for a neuron to be receptive. When a delayed rule is triggered, an initial start rule will consume the required spikes alongside the o flag, effectively closing the compartment mathematically. Simultaneously, it will generate d copies of a timer object (a_j) to initiate the clock, along with temporary processing flags (c and ν_j).

To ensure precise synchronization, the compartment will be governed by a strict four-block sequential execution strategy, defined as $\delta_i = B_1B_2B_3B_4$. Block B_1, utilizing a choice strategy, will handle either the initiation of new rules or the final emission of spikes for completed rules. Block B_2, applied in a maximal parallel manner, will simulate the rejection of incoming signals.

The firing rule will consume the temporary flags, emit the p spikes to the target compartments, and regenerate the f and o flags, thereby restoring the initial f^2o configuration and reopening the neuron. By enforcing this state-machine mapping, future iterations of the kPWorkbench tool will replicate the complex, time-delayed biological behaviors of SN P systems using only fundamental mul-

tiset rewriting rules, expanding its utility as a comprehensive development environment for membrane computing.

Furthermore, alongside expanding the theoretical scope of the pipeline, an immediate direction for future research involves a rigorous empirical evaluation of the translation framework's efficiency and computational limits. While the current architecture successfully normalizes and translates large-scale networks, the injection of strict sequential execution strategies and maximal parallel blocks introduces inherent computational overhead during the simulation phase.

Acknowledgment We would like to thank the reviewers for their detailed and useful comments that allowed the quality of the paper to be improved.

A XML Instance: 4-Channel Bitonic Sorter

The following Extensible Markup Language (XML) serialization represents the unified intermediate schema generated by the translation pipeline for the 4-channel bitonic sorting network (originally authored in MeCoSim via P-Lingua). This normalized structure acts as the source for the final kP-Lingua code generation.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<snpSystem type="plingua">
  <alphabet>
    <spike symbol="a" />
  </alphabet>
  <neuronTypes>
    <neuronType id="t_N0">
      <rules>
        <rule regex="a+" consumed="1" produced="1" delay="0"
        type="spiking" id="r1" />
      </rules>
    </neuronType>
    <neuronType id="t_N4">
      <rules>
        <rule regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" id="r1" />
        <rule regex="1a" consumed="1" produced="0" delay="0"
        type="forgetting" id="r2" />
      </rules>
    </neuronType>
    <neuronType id="t_N5">
      <rules>
        <rule regex="2a" consumed="2" produced="0" delay="0"
        type="forgetting" id="r1" />
        <rule regex="1a" consumed="1" produced="1" delay="0"
        type="spiking" id="r2" />
```

```
      </rules>
    </neuronType>
    <!-- Subsequent comparator neuron types (t_N6 through t_N23)
         follow the identical MAX/MIN structural logic. -->
    <neuronType id="t_N24">
      <rules />
    </neuronType>
  </neuronTypes>
  <neurons>
    <neuron id="N0" type="t_N0" initialSpikes="8" />
    <neuron id="N1" type="t_N1" initialSpikes="5" />
    <neuron id="N2" type="t_N2" initialSpikes="1" />
    <neuron id="N3" type="t_N3" initialSpikes="3" />
    <neuron id="N4" type="t_N4" initialSpikes="0" />
    <neuron id="N5" type="t_N5" initialSpikes="0" />
    <!-- Internal cascaded neurons N6 to N23 omitted for brevity -->
    <neuron id="N24" type="t_N24" initialSpikes="0" />
    <neuron id="N25" type="t_N25" initialSpikes="0" />
    <neuron id="N26" type="t_N26" initialSpikes="0" />
    <neuron id="N27" type="t_N27" initialSpikes="0" />
  </neurons>
  <synapses>
    <synapse source="N0" target="N4" />
    <synapse source="N0" target="N5" />
    <synapse source="N1" target="N4" />
    <synapse source="N1" target="N5" />
    <synapse source="N2" target="N6" />
    <synapse source="N2" target="N7" />
    <synapse source="N3" target="N6" />
    <synapse source="N3" target="N7" />
    <synapse source="N4" target="N8" />
    <synapse source="N4" target="N9" />
    <synapse source="N5" target="N9" />
    <!-- Remaining internal synapses omitted for brevity -->
    <synapse source="N20" target="N24" />
    <synapse source="N20" target="N25" />
    <synapse source="N21" target="N25" />
    <synapse source="N22" target="N26" />
    <synapse source="N22" target="N27" />
    <synapse source="N23" target="N27" />
  </synapses>
</snpSystem>
```

References

1. Cabarle, F., Adorna, H., Pérez-Jiménez, M., Song, T.: Spiking neural p systems with structural plasticity. Neural Computing and Applications **26** (02 2015). https://doi.org/10.1007/s00521-015-1857-4
2. Cabarle, F.G.C., Adorna, H.N., Jiang, M., Zeng, X.: Spiking neural p systems with scheduled synapses. IEEE Transactions on NanoBioscience **16**(8), 792–801 (2017). https://doi.org/10.1109/tnb.2017.2762580
3. Chen, Y., Chen, Y., Zhang, G., Paul, P., Wu, T., Zhang, X., Rong, H., Ma, X.: A survey of learning spiking neural p systems and a novel instance. International Journal of Unconventional Computing **16**(2–3), 173–200 (2021)
4. 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.: Snapse: A visual tool for spiking neural P systems. Processes **9**(1) (2021). https://doi.org/10.3390/pr9010072
5. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - Version I. In: 11^{th} Int. Conference on Membrane Computing. pp. 97–124. Fénix Editora (2013), `http://www.gcn.us.es/files/11bwmc/097_gheorghe_ipate.pdf`
6. Gheorghe, M., Ipate, F., Kannan, K., Konur, S., Kuppusamy, L., Lefticaru, R., Mahendran, A., Niculescu, M.I.: Spiking neural P systems and kernel P systems. Journal of Membrane Computing **7**(4), 437–459 (2025)
7. Guo, P., Quan, C., Ye, L.: Upsimulator: A general P system simulator. Knowledge-Based Systems **170**, 20–25 (2019). https://doi.org/10.1016/j.knosys.2019.01.013
8. Iacob, Ş., Ipate, F., Niculescu, I.M., Konur, S.: Some new developments in kplingua. Journal of Membrane Computing (2026). https://doi.org/10.1007/s41965-026-00221-7, `https://doi.org/10.1007/s41965-026-00221-7`
9. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. Fundamenta Informaticae **134**(1–2), 97–110 (2014)
10. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology **4**(1), 83–92 (2015). https://doi.org/10.1021/sb500134w
11. Konur, S., Mierlă, L., Ipate, F., Gheorghe, M.: kpworkbench: A software suite for membrane systems. SoftwareX **5**, 19–24 (2016). https://doi.org/10.1016/j.softx.2016.03.001
12. Leporati, A., Mauri, G., Zandron, C.: Spiking neural p systems: main ideas and results. Natural Computing **21**(4), 629–649 (2022). https://doi.org/10.1007/s11047-022-09917-y
13. Liu, Y., Chen, Y., Paul, P., Fan, S., Ma, X., Zhang, G.: A review of power system fault diagnosis with spiking neural p systems. Applied Sciences **11**(10), 4376 (2021)
14. Liu, Y., Zhao, Y.: Spiking neural p systems with lateral inhibition. Neural Networks (08 2023). https://doi.org/10.1016/j.neunet.2023.08.013
15. Lv, Z., Bao, T., Zhou, N., Peng, H., Huang, A., Riscos-Núñez, A., Pérez-Jiménez, M.: Spiking neural p systems with extended channel rules. International Journal of Neural Systems **31**(1), 2050049 (2021). https://doi.org/10.1142/s0129065720500490
16. Macababayao, I., Cabarle, F., de la Cruz, R.T., Zeng, X.: Normal forms for spiking neural p systems and some of its variants. Information Sciences **595** (03 2022). https://doi.org/10.1016/j.ins.2022.03.002

17. Pan, L., Păun, G., Pérrez-Jiménez, M.: Spiking neural p systems with neuron division and budding. Science China Information Sciences **54**(8), 1596–1607 (2011)
18. Pan, L., Păun, G.: Spiking neural P systems with anti-spikes. International Journal of Computers, Communications & Control **IV**(3), 273–282 (2009)
19. Pan, L., Wu, T., Zhang, Z.: A bibliography of spiking neural p systems. Bulletin of the International Membrane Computing Society **1**(1), 63–78 (2016)
20. Pan, L., Wang, J., Hoogeboom, H.: Spiking neural p systems with astrocytes. Neural Computation **24**, 805–825 (03 2012). https://doi.org/10.1162/NECO˙a˙00238
21. Pan, L., Zeng, X., Zhang, X., Jiang, Y.: Spiking neural p systems with weighted synapses. Neural Processing Letters **35**, 13–27 (02 2012). https://doi.org/10.1007/s11063-011-9201-1
22. Paul, P., Sosík, P., Ciencialová, L.: A survey on learning models of spiking neural membrane systems and spiking neural networks. arXiv preprint arXiv:2403.18609 (2024)
23. Peng, H., Bao, T., Luo, X., Wang, J., Song, X., Riscos-Núñez, A., Pérez-Jiménez, M.: Dendrite p systems. Neural Networks **127**, 110–120 (jul 2020). https://doi.org/10.1016/j.neunet.2020.04.014
24. Peng, H., Lv, Z., Li, B., Luo, X., Wang, J., Song, X., Wang, T., Pérez-Jiménez, M., Riscos-Núñez, A.: Nonlinear spiking neural p systems. International Journal of Neural Systems **30**(10), 2050008 (2020). https://doi.org/10.1142/s0129065720500088
25. Peng, H., Wang, J.: Coupled neural p systems. IEEE Transactions on Neural Networks and Learning Systems **30**(6), 1672–1682 (2019). https://doi.org/10.1109/TNNLS.2018.2872583
26. Peng, H., Wang, J.: Advanced Spiking Neural P Systems: Models and Applications. Springer Nature (2024). https://doi.org/10.1007/978-981-99-6379-9
27. Peng, H., Wang, J., Pérez-Jiménez, M., Wang, H., Shao, J., Wang, T.: Fuzzy reasoning spiking neural p system for fault diagnosis. Information Sciences **235**, 106–118 (2013). https://doi.org/10.1016/j.ins.2013.02.002
28. Peng, H., Li, B., Wang, J., Xiaoxiao, S., Wang, T., Valencia-Cabrera, L., Pérez-Hurtado, I., Riscos-Núñez, A., Pérez-Jiménez, M.: Spiking neural p systems with inhibitory rules. Knowledge-Based Systems **188**, 105064 (09 2019). https://doi.org/10.1016/j.knosys.2019.105064
29. Peng, H., Wang, J., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Dynamic threshold neural p systems. Knowledge-Based Systems **163**, 875–884 (2019). https://doi.org/https://doi.org/10.1016/j.knosys.2018.10.016, https://www.sciencedirect.com/science/article/pii/S0950705118305021
30. Peng, H., Yang, J., Wang, J., Wang, T., Sun, Z., Song, X., Luo, X., Huang, X.: Spiking neural p systems with multiple channels. Neural Networks **95**, 66–71 (2017). https://doi.org/https://doi.org/10.1016/j.neunet.2017.08.003, https://www.sciencedirect.com/science/article/pii/S0893608017301843
31. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
32. Pérez-Hurtado, I., Orellana-Martín, D., del Amor, M.M., Valencia-Cabrera, L., Riscos-Núñez, A.: A new P-lingua toolkit for agile development in membrane computing. Information Sciences **587**, 1–22 (2022). https://doi.org/10.1016/j.ins.2021.12.003
33. Rong, H., Wu, T., Pan, L., Zhang, G.: Spiking neural P systems: Theoretical results and applications, Lecture Notes in Computer Science, vol. 11270, pp. 256–268. Springer, Cham (2018)

34. Song, T., Gong, F., Liu, X., Zhao, Y., Zhang, X.: Spiking neural p systems with white hole neurons. IEEE Transactions on NanoBioscience **15**, 1–1 (10 2016). https://doi.org/10.1109/TNB.2016.2598879
35. Song, T., Pan, L.: Spiking neural p systems with request rules. Neurocomputing **193** (02 2016). https://doi.org/10.1016/j.neucom.2016.02.023
36. Song, T., Pan, L., Păun, G.: Spiking neural p systems with rules on synapses. Theoretical Computer Science **529** (04 2014). https://doi.org/10.1016/j.tcs.2014.01.001
37. Song, T., Rodríguez-Patón, A., Zheng, P., Zeng, X.: Spiking neural p systems with colored spikes. IEEE Transactions on Cognitive and Developmental Systems **PP**, 1–1 (12 2017). https://doi.org/10.1109/TCDS.2017.2785332
38. Valencia-Cabrera, L., Song, B.: Tissue P systems with promoter simulation with mecosim and P-lingua framework. Journal of Membrane Computing **2** (2020). https://doi.org/10.1007/s41965-020-00037-z
39. Wang, J., Hoogeboom, H., Pan, L., Păun, G., Pérez-Jiménez, M.: Spiking neural p systems with weights. Neural Computation **22**, 2615–2646 (10 2010). https://doi.org/10.1162/NECO˙a˙00022
40. Wu, T., Pan, L., Tan, K.: Numerical spiking neural p systems. IEEE Transactions on Neural Networks and Learning Systems **32**(6), 2816–2829 (2021). https://doi.org/10.1109/TNNLS.2020.3008546
41. Wu, T., Paun, A., Zhang, Z., Pan, L.: Spiking neural p systems with polarizations. IEEE Transactions on Neural Networks and Learning Systems **PP**, 1–12; (08 2017). https://doi.org/10.1109/TNNLS.2017.2726119
42. Zhang, G., Verlan, S., Wu, T., Cabarle, F., Xue, J., Orellan-Martín, D., Dong, J., Valencia-Cabrera, L., Pérrez-Jiménez, M.: Spiking Neural P Systems: Theory, Applications and Implementations. Springer Nature (2024). https://doi.org/10.1007/978-3-031-48762-0
43. Zhang, G., Zhang, X., Rong, H., Paul, P., Zhu, M., Neri, F., Ong, Y.S.: A layered spiking neural system for classification problems. International Journal of Neural Systems **32**(08), 2250023 (2022)
44. Zhang, X., Pan, L., Păun, A.: On the universality of axon p systems. IEEE Transactions on Neural Networks and Learning Systems **26**(11), 2816–2829 (2015). https://doi.org/10.1109/TNNLS.2015.2398516