A verification pipeline for neuronal archetypes

Thi-Thuy-Duong Pham[1,*], Elisabetta De Maria[1] and Robert De Simone[2]

[1]I3S, CNRS, Université Côte d'Azur, Sophia Antipolis, France
[1]I3S, CNRS, Université Côte d'Azur, Sophia Antipolis, France
[2]Centre Inria d'Université Côte d'Azur, Sophia Antipolis, France

Abstract

Spiking Neural Networks (SNNs) are considered to be more biologically realistic models of neural computation than traditional artificial neural networks, as they encode and process information through the timing of discrete spikes rather than continuous activation values. Within this framework, neuronal archetypes represent canonical connectivity patterns that serve as fundamental building blocks for constructing larger and more complex neural circuits. Understanding the dynamical properties of these archetypes is therefore a crucial first step toward explaining and predicting the behaviour of the networks that emerge from their composition.

In this work, we propose a formal verification framework for analysing the dynamics of neuronal archetypes through model checking. Our main contribution is the definition of a dedicated domain-specific language, named NASL (Neuronal Archetype Specification Language), to represent archetype structures and Leaky Integrate-and-Fire (LIF) neuron parameters, together with `snn_mc`, an effective software tool for the automatic translation and expansion of NASL specifications into discrete-time NuSMV models. Relevant dynamical properties are expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) and verified exhaustively using the NuSMV model checker, which produces counterexample traces whenever a property is violated. Through representative archetype case studies, we demonstrate how this approach provides rigorous guarantees about archetype behaviour and supports the systematic study of biologically inspired neural circuits. Finally, we discuss the extension of this framework toward the verification of archetype compositions, paving the way for the formal analysis of larger spiking neural networks built from these elementary structures.

Keywords

Spiking Neural Networks, Neuronal Archetypes, Biological Neural Networks, Model Checking, Dynamic Properties

1. Introduction

Understanding the human brain remains one of the most profound challenges in science. Biological neural networks exhibit highly complex dynamics that are difficult to analyse through observation or simulation alone: experimental manipulation of biological neural circuits is often infeasible or ethically constrained, and exhaustive simulation of all possible behaviours of a network rapidly becomes intractable as its size grows. Formal modelling and verification techniques offer a rigorous alternative, providing a mathematical framework for abstracting neural systems and exhaustively reasoning about their dynamics, rather than relying on a finite set of simulated trajectories [1]. Symbolic techniques in particular have proved effective at taming the state-space explosion that exhaustive verification would otherwise face, even for systems with very large numbers of reachable states [2].

A central obstacle to applying formal methods to neural systems lies in the trade-off between biological realism and computational tractability. Highly detailed models, such as the Hodgkin–Huxley model [3], capture a wide range of biological behaviours but are computationally expensive, which limits their use in formal verification settings. Simplified spiking neuron models, in particular the Leaky Integrate-and-Fire (LIF) model [4, 5], retain the essential temporal dynamics of spiking neural networks

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy
*Corresponding author.
thi-thuy-duong.pham@etu.univ-cotedazur.fr (T. Pham); elisabetta.de-maria@univ-cotedazur.fr (E. D. Maria); robert.de-simone@inria.fr (R. D. Simone)
https://github.com/thuyduong275 (T. Pham)
0009-0006-4915-7465 (T. Pham); 0000-0001-7116-9629 (E. D. Maria); 0000-0002-3123-7591 (R. D. Simone)

(SNNs) while remaining tractable enough to be encoded as finite-state systems, making them well suited to model checking.

Within the broader landscape of neural network models, SNNs are generally regarded as the third generation, following the McCulloch–Pitts neuron, in which units act as simple threshold logic gates [6], and continuous-activation models such as the multi-layer perceptron [7], whose outputs represent firing rates rather than discrete spikes. Unlike these earlier generations, SNNs incorporate the precise timing of individual spikes as an essential part of how information is processed [8, 9], which is what makes them closer to biological neural systems and the focus of this work.

Rather than directly tackling large, intractable neural networks, this work focuses on *neuronal archetypes* [10, 11]: small, recurrent micro-circuits that represent canonical connectivity patterns commonly found in biological neural systems, such as negative feedback loops and contralateral inhibition. Neuronal archetypes act as fundamental building blocks from which larger and more elaborate neural structures are composed. Establishing rigorous guarantees on the dynamical behaviour of individual archetypes is therefore a necessary first step toward understanding, and eventually verifying, the behaviour of the larger networks that emerge from their composition.

In this paper, we propose a formal verification framework for the automatic analysis of neuronal archetypes through model checking, implemented in the tool `snn_mc`. Archetype topologies and LIF neuron parameters are specified using NASL (Neuronal Archetype Specification Language), a dedicated domain-specific language, automatically translated into a discrete-time NuSMV [12] model. Relevant dynamical properties, such as guaranteed oscillation or mutual exclusion between competing neurons, are expressed in Computation Tree Logic (CTL) [13] and Linear Temporal Logic (LTL) [14], and verified exhaustively using the NuSMV model checker, which produces a counterexample trace whenever a property is violated. This pipeline allows archetype behaviour to be specified, modelled, and verified without manual intervention at the model-checking level, and is illustrated through a representative archetype case study.

Several works have applied formal verification techniques to spiking neural networks. Some studies map spiking neural P systems into timed automata to analyse their temporal properties [15]; others use timed automata to model LIF neural networks and verify system behaviours using temporal logic formulas [16]. De Maria et al. [10] propose a formal framework for modelling and verifying neuronal archetypes using Lustre, a synchronous language, to encode neurons and neuronal graphs and to express temporal properties concerning their dynamic behaviour, subsequently verified by model checkers. Beyond deterministic models, Yao et al. [17] propose a formal framework for modelling and verifying probabilistic SNNs, in which synaptic transmission and spike generation are governed by stochastic processes, combining probabilistic model checking with a biologically grounded neuron model to verify quantitative properties such as the probability of a neuron firing within a given time window. Our work is currently committed to temporal modelling in the CTL/LTL/NuSMV setting (discrete step sequences without physical time measures or probabilistic transitions); the same DSL-driven methodology should nevertheless be relevant to lift toward timed and stochastic verification frameworks.

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on the LIF neuron model, temporal logics, the NuSMV model checker, and neuronal archetypes. Section 3 presents the proposed verification pipeline, from the DSL specification to NuSMV model generation, property generation, and verification. Section 4 illustrates this pipeline on a representative negative-loop case study and surveys verification cost across the full archetype catalogue. Finally, Section 5 concludes the paper and discusses the extension of this framework toward the verification of archetype compositions.

2. Preliminaries

This section introduces the biological and theoretical background needed throughout the paper.

2.1. Leaky Integrate-and-Fire neuron model

The Leaky Integrate-and-Fire (LIF) model is a simplified computational model that captures the essential behaviour of biological neurons while remaining computationally efficient [4, 5]. A neuron accumulates excitatory and inhibitory post-synaptic potentials through spatial summation (across synapses) and temporal summation (over time); its membrane potential also decays through a leakage effect, and a spike is emitted whenever the potential exceeds a threshold [10, 5, 18].

A LIF is represented as a weighted directed graph, (V, E, w), where V is the set of neurons, $E \subseteq V \times V$ represents synaptic connections, and $w : E \to \mathbb{Z}$ assigns an integer synaptic strength to each edge, measured in membrane-potential units within $[0, P_{max}]$: a positive weight corresponds to excitation, a negative weight to inhibition [10].

Each neuron is characterized by a tuple (τ, r, p, y), where τ is the firing threshold, r is the leak factor, $p(t)$ the membrane potential at time t, and $y(t)$ the neuron output [4, 10]. The membrane potential follows the recursive update (Definition 1)

$$p(t) = \begin{cases} \sum_{i=1}^{m} w_i x_i(t), & \text{if } p(t-1) > \tau, \\ \sum_{i=1}^{m} w_i x_i(t) + r \cdot p(t-1), & \text{otherwise,} \end{cases}$$

with $x_i(t) \in \{0, 1\}$ and reset to zero after a spike. Equivalently, $p(t) = \sum_{e=0}^{\sigma} r^e \sum_{i=1}^{m} w_i x_i(t-e)$ expands the same dynamics over a window of length σ [10]; the NuSMV encoder uses the recursive form (Section 3).

2.2. Neuronal archetypes

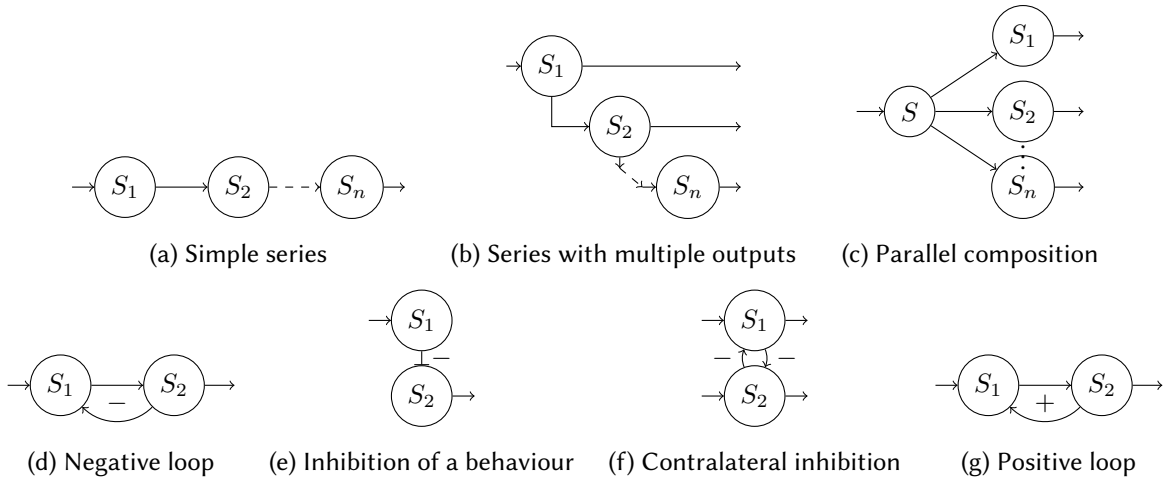
Neuronal archetypes are elementary neuronal circuits composed of a small number of neurons, each fulfilling a specific computational function [10, 11]. The central hypothesis is that any neuronal micro-circuit, regardless of its size or topological complexity, can be reduced to one of a finite set of such archetypes, which can further be coupled (by connecting outputs to inputs or by nesting) to form more elaborate functional structures [10, 19, 11]. We consider the six fundamental motifs from De Maria et al. [10] illustrated in Figure 1(a–f), together with a seventh positive feedback loop extension (g): *simple series*, a chain where each neuron receives the output of its predecessor; *series with multiple outputs*, a series whose every neuron's output is observed at each time step; *parallel composition*, a set of neurons driven by a single common input neuron; *negative loop*, two neurons where the first excites the second and the second inhibits the first, producing oscillatory behaviour; *inhibition of a behaviour*, two neurons where the first suppresses the activity of the second; *contralateral inhibition*, two or more neurons that mutually inhibit one another, giving a winner-takes-all dynamic; and *positive loop*, an excitatory ring that can sustain reverberatory activity when the loop gain exceeds threshold.

These archetypes are biologically inspired logical operators, easily adjustable by tuning a small number of parameters, and they constitute the basis of typical instances of neuronal information processing [10].

2.3. Temporal logic and model checking

The behaviour of a discretised LIF network can be represented as a transition system [1], defined as the tuple $T = (S, S_0, \Sigma, \to, AP, L)$, where S is the finite set of reachable states, $S_0 \subseteq S$ is the set of initial states, Σ is the set of input symbols (or actions), $\to \subseteq S \times \Sigma \times S$ is the transition relation, AP is the set of atomic propositions, and $L : S \to 2^{AP}$ is the labelling function. In our model, each state represents the membrane potentials and spike emission statuses of all neurons at a given discrete time instant, while each transition corresponds to one evolution step according to the recursive Definition 1 LIF encoding described in Section 2.1. Since P_{max} and the integer synaptic weights are bounded, the resulting state space is finite, making the system amenable to automatic verification [20, 19].

Two major temporal logics are used to express dynamical properties [1]: Computation Tree Logic (CTL) [13], which interprets formulas over the branching-time structure of computation by pairing



(a) Simple series (b) Series with multiple outputs (c) Parallel composition

(d) Negative loop (e) Inhibition of a behaviour (f) Contralateral inhibition (g) Positive loop

Figure 1: The six fundamental neuronal archetypes (a–f), adapted from [10], together with the positive feedback loop archetype (g) proposed in this work. Circles denote neurons and arrows denote synaptic connections. A connection labelled $+$ is excitatory (activation), while a connection labelled $-$ is inhibitory (inhibition).

each temporal operator (X, F, G, U) with a path quantifier A (all paths) or E (some path); Linear Temporal Logic (LTL) [14], evaluated over a single infinite execution without explicit path quantifiers.

Model checking automatically verifies whether a transition system T satisfies a formula φ, written $T \models \varphi$, by exhaustively exploring all reachable states. When a property does not hold, the model checker produces a counterexample trace witnessing the violation.

2.4. NuSMV

NuSMV [12] is a widely used symbolic model checker built on Binary Decision Diagrams (BDDs), supporting the verification of finite-state systems against specifications written in both CTL and LTL. A NuSMV model is described in a dedicated input language organized into modules, each declaring a set of state variables, an initial condition, and a transition relation that updates these variables at every discrete time step; properties to be verified are declared separately, using the CTLSPEC and LTLSPEC keywords. This modular structure makes NuSMV well suited to encoding the discrete states, recursive membrane potential equations, and threshold-triggered transitions of the LIF neuron model introduced in Section 2.1.

3. Verification pipeline for spiking neural network models

To practically evaluate the theoretical framework of Section 2, we developed `snn_mc`, an automated tool that implements a linear verification pipeline for spiking neural networks, bridging *formal modelling*, *DSL compilation*, and *model checking*. The tool takes a high-level archetype description written in NASL, our domain-specific language, compiles it into a discrete-time NuSMV model together with temporal specifications (CTLSPEC/LTLSPEC), invokes the NuSMV model checker (Section 2.4), and summarises the results. This workflow reflects the formal approach of [10, 20, 19], and the whole process is fully automated behind a single command-line invocation.

3.1. System overview and execution workflow

The system is designed as a two-stage pipeline, following the classical formal verification workflow *system specification* → *model construction* → *property verification*:

- **Stage 1 — Model Generation:** A NASL specification describes the structure and dynamics of a neural network; it is compiled into a finite-state transition system encoded in the `.smv` format accepted by NuSMV [12].

- **Stage 2 — Model Checking:** The generated model is executed in NuSMV to verify temporal logic properties expressed in CTL or LTL. NuSMV reports whether each property holds and, when it does not, produces a counterexample trace for analysis.

All stages are orchestrated by `main()` in `snn_mc/cli.py`, which is the unique entry point. Each module stays passive when imported and is only invoked by this orchestration layer. From the user's perspective, the entire process is triggered by a single command, `python -m snn_mc run <file.dsl> -out <dir>`, which (1) reads the NASL file, (2) compiles it into a formal NuSMV model and property file, (3) executes the combined model in NuSMV (Section 2.4), and (4) collects and summarises the verification results, including the number of satisfied and violated properties and detailed logs for debugging. This automation removes any need for manual interaction with the model checker. The complete source code of `snn_mc`, including the NASL grammar, the example archetype specifications, and the case study presented in Section 4, is publicly available at https://github.com/thuyduong275/neuro-archetype-verification-pipeline.

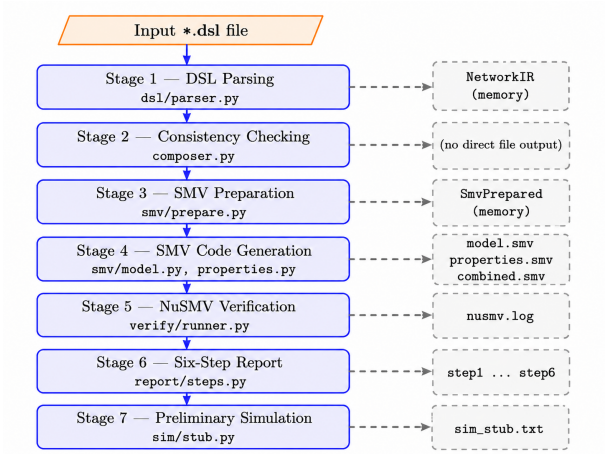


Figure 2: Processing workflow of the `snn_mc` system: from the input NASL file to the output artefacts. Each numbered stage corresponds to one step in the pipeline.

Figure 2 illustrates the complete processing workflow from the input `.dsl` file to the output artefacts written on disk. Solid arrows represent the main control flow; dashed arrows represent intermediate data structures or files on disk. The following subsections describe the compilation stages in detail; the end-to-end execution process, including reporting and simulation support, is summarised in Section 3.3.

3.2. Compilation from DSL to formal model

The transformation from NASL to a formal model is the core component of the system, since it directly impacts the correctness and expressiveness of the verification process.

3.2.1. Neural network representation

The internal representation must capture both topology and LIF dynamics in a form suitable for NuSMV code generation. The neural network is stored as a directed graph: nodes represent neurons and external inputs, and edges represent weighted synaptic connections, where a positive weight means excitation and a negative weight means inhibition, consistent with the weighted-graph definition of Section 2.1. Beyond individual edges, the system introduces *archetypes*, the canonical structural motifs that bridge these biological patterns with the automatic property templates emitted later in the pipeline.

3.2.2. DSL parsing and consistency checking

The parser in `snn_mc/dsl/parser.py` reads one NASL statement per line (`#` starts a comment), expands `include` directives, and builds a `NetworkIR` object containing neurons, `Edge` objects, LIF

parameter sets, input schedules, composition declarations, user `spec` lines, and explicit `block` archetype instances. When the parser encounters `block <kind> key=value ...`, it looks up `BLOCK_REGISTRY` and calls `apply_block()` on the matching archetype class, which expands the macro into neurons and edges at parse time. Table 1 summarises the core NASL statements. After parsing, `compose()` in `snn_mc/composer.py` performs semantic validation without changing the topology: every `input=` in a block must refer to a declared input or an already existing neuron, so chaining `block simple_series` before `block negative_loop input=c4` works as expected. This early validation catches wiring errors at the NASL level, before any NuSMV text is generated, which avoids opaque SMV syntax failures during verification later on. Listing 1 shows the parser's entry point and the block-expansion call it triggers.

Table 1
Core statements of NASL, the `snn_mc` domain-specific language.

Statement	Meaning
`include <file>`	Inline another NASL file
`horizon <int>`	Clock `t` ranges `0..horizon`
`input <name>`	External Boolean stimulus
`schedule <in> values ...`	Fixed input prefix (then nondeterministic)
`neuron_params <name> ...`	LIF bundle $(\tau, w_{exc}, w_{inh}, S, R, P_{max})$
`block <kind> key=value`	Archetype macro (neurons + edges)
`edge / exc / inh`	Manual weighted synapse
`compose seq|par ...`	Explicit composition between neurons
`spec CTLSPEC|LTLSPEC ...`	User-provided temporal property

Listing 1: Parser entry point and block expansion.

```
# block <kind> key=value ... -> BLOCK_REGISTRY[kind].apply_block(...)
ir = parse_file(dsl_path, override_n=override_n) # -> NetworkIR
ir = composer.compose(ir) # semantic validation only
```

3.2.3. Identifier sanitisation and SMV preparation

`prepare_ir()` in `snn_mc/smv/prepare.py` produces a NuSMV-safe snapshot, `SmvPrepared`, that is shared by all code generators. It sanitises every neuron and input name via `sanitize_identifier()` to avoid clashes with NuSMV reserved keywords, and propagates the renames to edges, schedules, compositions, and user spec text. Edges are grouped per neuron into excitatory (`exc_srcs`) and inhibitory (`inh_srcs`) source lists, reflecting spatial summation at each synapse. Explicit archetypes from NASL `block` lines are merged with graph-detected instances from `detect_archetypes`, deduplicated by kind and node set. Because the snapshot is a single immutable object, the model and property emitters that come after it always refer to the same renamed graph. Identifier sanitisation and the merged archetype list are produced once and shared by all subsequent emitters.

3.2.4. Automatic archetype detection

`detect_archetypes()` in `snn_mc/archetypes/__init__.py` automatically identifies structural motifs from the network graph when users define neuron connections manually instead of using `block` macros. It constructs a `GraphIndex` representation and applies a sequence of detection heuristics following a fixed `DETECTION_ORDER` (more specific patterns before generic ones). Detected motifs are mapped to the same verification property templates used by explicitly declared `block` macros. Seven block types are registered in `BLOCK_REGISTRY`, including `positive_loop` and `series_multiple_outputs` as practical extensions beyond the six motifs of Fig. 1.

3.2.5. Automatic property generation

`emit_properties_block()` in `snn_mc/smv/properties.py` emits temporal logic specifications that characterise archetype behaviour, so the user does not have to write every formula by hand. It always appends specs in a stable four-block order: (1) baseline LIF invariants per neuron (reset on spike, P within bounds, spike–threshold equivalence, fixed leak factor); (2) composition properties from explicit or inferred `compose` declarations; (3) archetype templates from `archetype_specs()` for each entry in `arch_list`; and (4) the user's verbatim `spec` lines.

Composition in the current tool. NASL supports `compose sequential|parallel` among named neurons, and archetype blocks attach the corresponding composition to the neurons of a *single* motif (e.g. chain edges in a series, fan-out in parallel composition). Chaining across macros is allowed by wiring one block's output into another's `input=` (as when a `simple_series` precedes a `negative_loop`). These constructs are compiled into one shared NuSMV model and generate CTL obligations (propagation along sequential edges; joint reachability / co-activation for parallel outputs). They do *not* yet provide assume–guarantee compositional proofs between separately verified archetypes; that stronger form of compositionality is left to future work (Section 5).

3.2.6. NuSMV model generation

`smv/model.py` emits `MODULE main` with the global clock, input schedules, per-edge weighted sums (`net_exc`, `net_inh`), and the neuron instances. `smv/lif_module.py` defines `MODULE lif_<pset>`, which implements the leak–integrate–fire update with a fixed leak factor R/S per parameter set, and `smv/combined.py` merges the model and properties into `combined.smv`. The `-emit-mode` flag selects either the full LIF encoding (`lif`, the default) or a discrete `simple_boolean` threshold mode that reduces state-space size when needed. The emitted modules are a direct symbolic encoding of the recursive Definition 1 LIF update of Section 2.1, which is what makes the transition system finite and amenable to the model checking framework of Section 2.3. The σ-window form recalled in Section 2.1 is an equivalent finite expansion of the same temporal summation; the NuSMV encoder prefers the recursive form so that each neuron stores only the membrane potential P and a fixed leak register, without input-history registers such as `in_hist_*`.

We consider three qualitatively different parameter sets, or presets, characterised by their firing threshold and leak factor: *slow*, *intermediate*, and *quick*. A *slow* neuron takes longer to reach its firing threshold, since its membrane potential leaks away more quickly and a higher threshold must be overcome before a spike is emitted; conversely, a *quick* neuron reaches its threshold more readily, as its potential decays more slowly and accumulates input more effectively, allowing it to spike sooner. *Intermediate* neurons exhibit a behaviour in between these two extremes. The three presets share the same underlying LIF dynamics described in Section 2.1 (leak $R/S{=}3/4$, $w_{exc}{=}3$, $P_{\max}{=}10$) and differ only in the threshold $\tau \in \{3, 6, 9\}$ for `quick`, `intermediate`, and `slow`, which makes them convenient building blocks for exploring how the timing of an archetype's behaviour depends on the responsiveness of its constituent neurons, as illustrated in the case study of Section 4.

Listing 2 shows the core of this encoding (illustrated for `params=quick`).

Listing 2: Recursive Definition 1 LIF update emitted by `lif_module.py`.

```
leak_term := (r_num * P) / S_leak;
raw_next := leak_term + input_sum;
spike := (P >= tau);
next(P) := case spike : 0; raw_next < 0 : 0;
        raw_next > Pmax : Pmax; TRUE : raw_next; esac;
```

3.3. End-to-end execution process

A single `python -m snn_mc run` invocation follows Fig. 2: parse the NASL file into `NetworkIR`, compose and validate block references, prepare a sanitised IR (identifiers, synapses, archetypes), emit `model.smv`/`properties.smv`/`combined.smv`, run NuSMV and summarise the log (including counterexample snippets when a property fails), write the six numbered step reports, and optionally emit a simulation stub unless `–skip-sim` is set.

Overall, this pipeline gives a systematic path from high-level neural network modelling to formal verification. By integrating NASL-based specification, automatic translation into NuSMV, and archetype-driven property generation, `snn_mc` supports rigorous analysis of dynamic SNN behaviours without manual model-checker interaction. A concrete application of this pipeline on a representative archetype is presented in Section 4.

4. Case study: negative feedback loop

This section runs the verification pipeline of Section 3 on the *negative feedback loop* archetype (Fig. 1(d)), one of the six fundamental motifs introduced in Section 2.2. Two neurons form a closed circuit: the first excites the second, and the second inhibits the first, a structure that can support alternating activity under suitable LIF parameters and input schedules [10]. We apply the pipeline from NASL specification through NuSMV model generation, automatic CTL/LTL property emission, and model checking, using the reference input `examples/negloop_only.dsl` (aligned with the catalogue survey of Section 4.8).

4.1. Motivation and circuit topology

The case-study network contains two neurons, `a` and `b`, and one Boolean input `stim`. The `negative_loop` block macro expands into three weighted edges: excitation from `stim` to `a`, excitation from `a` to `b`, and inhibition from `b` back to `a`. Both neurons use the `quick` parameter set; among the three presets, only `quick` allows the inhibited neuron to fire and supports alternating activity under stimulation (Section 4.6). A fixed input schedule drives `stim` for the first five discrete time steps; afterwards the input becomes nondeterministic, which lets NuSMV explore alternative stimulus continuations within the simulation horizon $t \in \{0, \ldots, 20\}$.

4.2. DSL specification

Listing 3 shows the complete NASL file. The `include` directive pulls in shared LIF parameter bundles from `neuron_base.dsl`; the `schedule` line fixes the stimulus prefix; and a single `block` line declares the archetype instance with explicit neuron names `A=a` and `B=b`.

Listing 3: Case-study NASL file (`examples/negloop_only.dsl`).

```
include neuron_base.dsl

input stim
schedule stim values TRUE FALSE TRUE TRUE FALSE

block negative_loop input=stim A=a B=b params=quick
```

The parser produces a `NetworkIR` with two neurons, three edges, and one explicit `negative_loop` archetype instance. We run with `–no-detect` so that graph-based detection does not overlay a `simple_series` on the excitatory chain; only the explicit block contributes archetype templates, matching the survey protocol.

4.3. Applying the pipeline

The entire case study is produced by one command:

```
python -m snn_mc run examples/negloop_only.dsl --no-detect --out runs/case_study_negloop
```

Following Fig. 2, the tool parses and validates the NASL file, prepares a sanitised `SmvPrepared` snapshot, emits `model.smv`, `properties.smv`, and `combined.smv`, invokes NuSMV, and writes the six numbered report files (`step1_diagram.md` through `step6_results.txt`). Both NuSMV and the tool return exit code 0: all fifteen specifications hold (Section 4.6).

4.4. Generated NuSMV model

Listing 4 excerpts the generated `MODULE main` and the `lif_quick` submodule. The global clock `t` ranges over $0..20$. Weighted synaptic inputs are computed per edge in the main module (`net_exc` / `net_inh` arguments). Each neuron stores only the membrane potential `P` and the fixed leak register `r_num`; the recursive Definition 1 update (Listing 2) needs no input-history registers.

Listing 4: Excerpt of generated `model.smv` (case study).

```
MODULE main
VAR
  t : 0..20;
  stim : boolean;
  a : lif_quick(((stim) ? 3 : 0), ((b.spike) ? -3 : 0), t);
  b : lif_quick(((a.spike) ? 3 : 0), 0, t);

MODULE lif_quick(net_exc, net_inh, t)
VAR P : 0..10; r_num : 0..4;
DEFINE
  Pmax := 10; tau := 3; S_leak := 4;
  input_sum := net_exc + net_inh;
  leak_term := (r_num * P) / S_leak;
  raw_next := leak_term + input_sum;
  spike := (P >= tau);
ASSIGN
  init(P) := 0; init(r_num) := 3;
  next(r_num) := r_num;
  next(P) := case spike : 0; raw_next < 0 : 0;
          raw_next > Pmax : Pmax; TRUE : raw_next; esac;
```

4.5. Generated verification properties

The property emitter concatenates two groups (Section 3): eight baseline LIF invariants (four per neuron: reset on spike, membrane bounds, spike–threshold equivalence, and fixed leak factor) and seven archetype templates for `negative_loop` on `[a,b]`: mutual exclusion, two reachability checks, two CTL feedback formulas, one *driven* oscillation LTL formula under continuous stimulation, and one LTL quiescence property. In total NuSMV checked **fifteen** specifications for this input (Table 2).

4.6. Verification results

The choice of parameter set is dictated by a structural condition for oscillation in the two-neuron negative loop: the downstream neuron can cross its threshold from a single presynaptic pulse only when $w_{exc} \geq \tau$. Among the three fixed neuron types this inequality holds only for the `quick` preset ($\tau = 3 \leq w_{exc} = 3$); for `intermediate` ($\tau = 6$) and `slow` ($\tau = 9$) we verified that $\mathbf{EF}\, b.spike$ is **false**: the inhibited neuron never reaches its threshold and the loop cannot oscillate at all. We therefore instantiate both neurons with `quick` (selected by grid search over the three built-in presets; Section 4.8).

Under `quick`, NuSMV verified **all fifteen** specifications. All eight baseline LIF invariants hold, together with the mutual exclusion property $\mathbf{AG}\,\neg(a.spike \wedge b.spike)$, the reachability of both neurons ($\mathbf{EF}\, a.spike$, $\mathbf{EF}\, b.spike$), the bidirectional feedback $\mathbf{AG}(a.spike \Rightarrow \mathbf{EF}\, b.spike)$ and $\mathbf{AG}(b.spike \Rightarrow$

$\mathbf{EF}\, a.spike$), the driven oscillation $\mathbf{G}\, stim \Rightarrow (\mathbf{GF}\, a.spike \wedge \mathbf{GF}\, b.spike)$, and the LTL quiescence property $\mathbf{G}\, \neg stim \Rightarrow \mathbf{G}\, (\neg a.spike \wedge \neg b.spike)$. The driven form replaces unconditional recurrent oscillation, which fails when `stim` may stay low after the scheduled prefix: the loop alternates while driven and becomes silent without input. A representative driven trace alternates `a` and `b` firing in successive steps, the characteristic back-and-forth of a negative feedback loop. Table 2 summarises the outcome.

Table 2
Verification results for `negloop_only.dsl` (horizon 20, `params=quick`, `-no-detect`): all fifteen specifications hold.

Property (abbrev.)	Logic	Result
Reset on spike / membrane bounds / semantic / fixed leak (a, b)	CTL	true (8)
Mutual exclusion $\neg(a.spike \wedge b.spike)$ always	CTL	true
Reachability $\mathbf{EF}\, a.spike$, $\mathbf{EF}\, b.spike$	CTL	true (2)
Feedback $a.spike \Rightarrow \mathbf{EF}\, b.spike$ and converse	CTL	true (2)
Driven oscillation $\mathbf{G}\, stim \Rightarrow (\mathbf{GF}\, a.spike \wedge \mathbf{GF}\, b.spike)$	LTL	true
Quiescence $\mathbf{G}\, \neg stim \Rightarrow \mathbf{G}(\neg a.spike \wedge \neg b.spike)$	LTL	true

4.7. Evaluation: model size and verification cost

Beyond the qualitative outcome, we report the quantitative cost of the analysis, since tractability is a central concern for symbolic model checking of LIF networks. All measurements use NuSMV 2.7.1 (win64) with BDD dynamic variable reordering enabled (the `-dynamic` flag passed by `snn_mc`).

The generated model is compact: two neurons, one Boolean input, and three weighted edges. Each neuron contributes only the membrane potential $\mathtt{P} \in [0, 10]$ and the fixed leak register `r_num`; together with the global clock $\mathtt{t} \in [0, 20]$ and `stim`, exhaustive reachability analysis yields **98** reachable states, represented by about 1,933 peak live BDD nodes, and NuSMV checks all fifteen CTL/LTL specifications in about **0.02 s** (Table 3). The same cost figures appear for `negative_loop` in the catalogue survey (Table 4).

Table 3
Verification cost of the case study: a naive σ-window expansion versus the recursive Definition 1 encoding emitted by `snn_mc`. Both runs use the same properties and NuSMV configuration (horizon 20, `params=quick`).

Metric	Naive (σ-window)	`snn_mc` (recursive)
Effective window / history registers	σ=8, 8 regs/neuron	none (`raw_next`)
History register range	$[-10, 10]$	n/a
Declared state space	$2^{87.2}$	compact (P, r_{num} only)
Reachable states	536	98
BDD nodes (allocated / peak live)	168,596	1,933 (peak live)
15-spec verification time	$>$ 300 s (timeout)	0.02 s

These data depend on the encoding choice in the model emitter (Listing 4). Table 3 compares a naive encoding that expands the analytic σ-window with worst-case history registers against the recursive Definition 1 module produced by `snn_mc`. Avoiding history registers shrinks the declared state space dramatically and reduces the BDD from 168,596 allocated nodes to about 1,933 peak live nodes, cutting verification time from a five-minute timeout to about 0.02 s without changing the intended leak–integrate–fire dynamics. The two formulations agree on the underlying membrane update; the recursive form simply stores P instead of an explicit input history.

Because the encoding is per neuron and the property set grows linearly with the number of archetype instances, the dominant cost is the BDD representation of the joint neuron state. The case study confirms

that, for the small motifs that make up the archetype catalogue, this cost stays well within interactive range, and that the recursive encoding is what keeps high-retention presets such as `quick` tractable.

4.8. Survey across the archetype catalogue

Beyond the detailed negative-loop walkthrough, we apply the same pipeline to **all seven** registered archetype kinds on the minimal reference files `examples/*_only.dsl`. Each run uses horizon $t \in \{0, \ldots, 20\}$, the `quick` neuron preset ($\tau = 3$, leak $R/S = 3/4$, $w_{exc} = 3$), and `-no-detect` so that only the explicit `block` contributes archetype templates. The preset was selected by a grid search over the three built-in presets of Section 3. Table 4 reports, for each motif, the number of specifications checked together with true/false outcomes and the computational cost under NuSMV 2.7.1 with BDD dynamic reordering (`-dynamic`): reachable states, peak live BDD nodes, and wall-clock time. With this calibration, **every** checked property holds on every motif. All seven networks verify in well under 0.1 s; the largest motif (`simple_series`, N=5, 35 specs) uses only a few thousand live BDD nodes.

Table 4
Catalogue survey (horizon 20, `params=quick`, `-no-detect`, NuSMV `-dynamic`). Specs = number of CTL/LTL specifications checked; True/False = verification outcome; Time is wall-clock for checking all emitted specifications.

Archetype	N	Specs	True	False	Reach.	BDD live	Time (s)
`simple_series`	5	35	35	0	318	4,290	0.09
`series_multiple_outputs`	2	14	14	0	96	1,644	0.02
`parallel_composition`	4	27	27	0	102	3,669	0.03
`negative_loop`	2	15	15	0	98	1,933	0.02
`positive_loop`	2	14	14	0	104	1,950	0.02
`contralateral_inhibition`	2	13	13	0	69	2,045	0.02
`inhibition_of_behavior`	2	14	14	0	258	1,972	0.02

Neuron presets: `quick` → `intermediate` → `slow`. Replacing `quick` by `intermediate` or `slow` does *not* break the baseline LIF invariants (reset, bounds, spike–threshold equivalence, fixed leak). What typically fails are downstream **reachability** and **propagation** claims, because a single excitatory pulse of weight 3 cannot cross τ=6 or τ=9. The grid search recorded, for example, `simple_series` at 28 true / 7 false and `parallel_composition` at 18 true / 9 false under both `intermediate` and `slow`, whereas under `quick` both motifs are fully successful. By contrast, `contralateral_inhibition` stays all-true for all three presets after the formula revision, so not every key property is preset-fragile. This is why the catalogue table above calibrates on `quick`.

The six fundamental motifs of De Maria et al. together with the positive-loop extension and the two primitive composition operators (`compose sequential|parallel`) are expressive enough to assemble small networks in the spirit of the archetype hypothesis, but they do not by themselves guarantee that large networks can be designed and verified compositionally: that still requires interface contracts and assume–guarantee reasoning (Section 5). Scaling beyond catalogue-sized motifs remains future work.

5. Conclusion and future work

We presented `snn_mc`, an automated verification pipeline that connects high-level archetype descriptions to exhaustive NuSMV analysis. A compact DSL specifies LIF parameters, connectivity, and archetype blocks; the tool compiles a discrete-time model, emits baseline LIF invariants and archetype-specific CTL/LTL templates, and reports verification outcomes together with counterexample traces whenever a property is refuted. The negative-loop case study (Section 4) illustrated the full pipeline on a representative motif whose fifteen specifications all hold under the calibrated `quick` preset. The

catalogue survey (Section 4.8, Table 4) further shows that, under the same preset, all seven registered motifs admit fully successful verification runs within interactive time.

Composition support (current scope). The tool already implements *primitive* composition: users may declare `compose sequential|parallel` among named neurons, and archetype `block` macros attach the corresponding sequential or parallel composition to the neurons of a single motif; chaining is possible by feeding one block's output into another's `input=` (e.g. a series into a negative loop). These declarations are compiled into the shared NuSMV `MODULE main` and yield CTL obligations (edge-wise propagation for sequential composition; joint reachability / co-activation for parallel composition). What is *not* yet supported is compositional verification in the strong sense: deducing global properties of a multi-archetype network from separately verified interface contracts of its parts [21].

We plan to extend the framework toward that form of *compositional* verification, coupling outputs of one archetype to inputs of another, or nesting archetypes, as envisaged in the archetype hypothesis of [10, 19], by defining explicit interface contracts and studying how CTL/LTL properties transfer, or fail to transfer, under composition, including deduplication when graph detection overlaps explicit blocks. A complementary challenge is *scalability*: as multi-archetype networks grow beyond catalogue-sized motifs, state-space and BDD explosion must be addressed through modular encodings, abstraction, or assume–guarantee reasoning between coupled micro-circuits.

Acknowledgments

This research was conducted during a Master's internship at the I3S Laboratory, Université Côte d'Azur. The first author gratefully acknowledges financial support from the EFELIA Scholarship Program (France).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, D. L. Dill, Symbolic model checking for sequential circuit verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13 (1994) 401–424. doi:10.1109/43.275352.

[3] A. L. Hodgkin, A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve, The Journal of Physiology 117 (1952) 500–544. doi:10.1113/jphysiol.1952.sp004764.

[4] L. Lapicque, Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation, Journal de Physiologie et de Pathologie Générale 9 (1907) 620–635.

[5] W. Gerstner, W. M. Kistler, Spiking Neuron Models: Single Neurons, Populations, Plasticity, Cambridge University Press, Cambridge, 2002. doi:10.1017/CBO9780511815706.

[6] W. S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics 5 (1943) 115–133. doi:10.1007/BF02478259.

[7] G. Cybenko, Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals and Systems 2 (1989) 303–314. doi:10.1007/BF02551274.

[8] W. Maass, Networks of spiking neurons: The third generation of neural network models, Neural Networks 10 (1997) 1659–1671. doi:10.1016/S0893-6080(97)00011-7.

[9] H. Paugam-Moisy, S. M. Bohte, Computing with spiking neuron networks, in: Handbook of Natural Computing, Springer, 2012, pp. 335–376. doi:10.1007/978-3-540-92910-9_10.

[10] E. De Maria, A. Bahrami, T. L'Yvonnet, A. Felty, D. Gaffé, A. Ressouche, F. Grammont, On the use of formal methods to model and verify neuronal archetypes, Frontiers of Computer Science 16 (2022) 163404. doi:10.1007/s11704-020-0029-6.

[11] E. De Maria, A. Muzy, D. Gaffé, A. Ressouche, F. Grammont, Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models, Research Report RR-8937, UCA, Inria ; UCA, I3S ; UCA, LEAT ; UCA, LJAD, 2016. URL: https://inria.hal.science/hal-01349019.

[12] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model verifier, in: Computer Aided Verification, CAV 1999, Springer, 1999, pp. 495–499. doi:10.1007/3-540-48683-6_44.

[13] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (1986) 244–263. doi:10.1145/5397.5399.

[14] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, FOCS 1977, IEEE, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[15] B. Aman, G. Ciobanu, Modelling and verification of weighted spiking neural systems, Theoretical Computer Science 623 (2016) 92–102. doi:10.1016/j.tcs.2015.11.005.

[16] E. De Maria, C. Di Giusto, Parameter learning for spiking neural networks modelled as timed automata, in: Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2018) – Volume 3: BIOINFORMATICS, SciTePress, 2018, pp. 17–28. doi:10.5220/0006530300170028.

[17] Z. Yao, E. De Maria, R. De Simone, Probabilistic spiking neural networks: Formal verification and simulation, in: Unconventional Computation and Natural Computation (UCNC 2025), volume 16364 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 100–114. doi:10.1007/978-3-032-15641-9_8.

[18] D. Purves, G. J. Augustine, D. Fitzpatrick, W. C. Hall, A.-S. LaMantia, J. O. McNamara, S. M. Williams (Eds.), Neuroscience, 3rd ed., Sinauer Associates, 2006.

[19] E. De Maria, T. L'Yvonnet, D. Gaffé, A. Ressouche, F. Grammont, Modelling and formal verification of neuronal archetypes coupling, in: Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics, ACM, 2017, pp. 3–10. doi:10.1145/3156346.3156348.

[20] E. De Maria, A. Muzy, D. Gaffé, A. Ressouche, F. Grammont, Verification of temporal properties of neuronal archetypes modeled as synchronous reactive systems, in: Hybrid Systems Biology, HSB 2016, Springer, 2016, pp. 97–112. doi:10.1007/978-3-319-47151-8_7.

[21] E. M. Clarke, D. E. Long, K. L. McMillan, Compositional model checking, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), IEEE Computer Society, 1989, pp. 353–362. doi:10.1109/LICS.1989.39190.