HAL
open science

# On the Use of Formal Methods to Model and Verify Neuronal Archetypes

Elisabetta de Maria, Abdorrahim Bahrami, Thibaud L'Yvonnet, Amy Felty, Daniel Gaffé, Annie Ressouche, Franck Grammont

## To cite this version:

Elisabetta de Maria, Abdorrahim Bahrami, Thibaud L'Yvonnet, Amy Felty, Daniel Gaffé, et al.. On the Use of Formal Methods to Model and Verify Neuronal Archetypes. Frontiers of Computer Science, 2021, 16, pp.28. hal-03053930

## HAL Id: hal-03053930
## https://hal.science/hal-03053930

Submitted on 11 Dec 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# On the Use of Formal Methods to Model and Verify Neuronal Archetypes

Elisabetta DE MARIA[1], Abdorrahim BAHRAMI[2], Thibaud L'YVONNET[3], Amy FELTY[2], Daniel GAFFÉ[4], Annie RESSOUCHE[3], and Franck GRAMMONT[5]

[1]Université Côte d'Azur, CNRS, I3S, 06903 Sophia Antipolis Cedex, France
[2]School of Electrical Engineering and Computer Science, University of Ottawa, Ontario, K1N 6N5, Canada
[3]Université Côte d'Azur, INRIA SAM, 06902 Sophia Antipolis Cedex, France
[4]Université Côte d'Azur, CNRS, LEAT, 06903 Sophia Antipolis cedex, France
[5]Université Côte d'Azur, CNRS, LJAD, 06108 Nice Cedex 02, France

December 11, 2020

**Abstract**

Having a formal model of neural networks can greatly help in understanding and verifying their properties, behavior, and response to external factors such as disease and medicine. In this paper, we adopt a formal model to represent neurons, some neuronal graphs, and their composition. Some specific neuronal graphs are known for having biologically relevant structures and behaviors and we call them archetypes. These archetypes are supposed to be the basis of typical instances of neuronal information processing. In this paper we study six fundamental archetypes (simple series, series with multiple outputs, parallel composition, negative loop, inhibition of a behavior, and contralateral inhibition), and we consider two ways to couple two archetypes: (i) connecting the output(s) of the first archetype to the input(s) of the second archetype and (ii) nesting the first archetype within the second one. We report and compare two key approaches to the formal modeling and verification of the proposed neuronal archetypes and some selected couplings. The first approach exploits the synchronous programming language Lustre to encode archetypes and their couplings, and to express properties concerning their dynamic behavior. These properties are verified thanks to the use of model checkers. The second approach relies on a theorem prover, the Coq Proof Assistant, to prove dynamic properties of neurons and archetypes.

***Keywords***— Neuronal Networks, Leaky Integrate and Fire Modeling, Synchronous Languages, Model Checking, Theorem Proving, Lustre, Coq, Formal Methods.

## 1 Introduction

In the last years, much attention has been directed towards the study of the structure and function of brain networks. This research is often grouped under the now well-known keyword of the (human) *connectome* project [1, 2, 3]. However, although the research in this field claims to do graph analysis of the connectome at micro and macro scales, it mainly focuses on macro scales and is mostly based on diffusion or functional MRI data in humans [4].

Here, we want to promote a more fundamental and formal approach to the study of specific neuronal micro-circuits, namely *neuronal archetypes*. From the biological point of view, the theory of neuronal archetypes postulates that the most primitive circuits that emerge during phylogenetic and ontogenetic evolution (i.e., in the first living systems and mostly in the mammal spinal cord and brain stem, or homolog in other phyla) are elementary circuits of a few neurons fulfilling a specific computational function (e.g., contralateral inhibition; see Section 4). The idea is that archetypes constitute the normalized form of potentially bigger and topologically more complicated neuronal circuits, but not

more complex, in the sense that they do not perform other computations than the reference archetype itself does. In other words, every micro-circuit, even with many neurons, can theoretically be reduced to one of the few existing archetypes. Archetypes can be coupled in different ways. If the resulting circuit does not perform a more complex function in terms of neuronal information processing, it means that it should theoretically be reducible to a smaller archetype. If a new specific function, biologically relevant, is identified, then it can be categorized as a new and bigger archetype. From the informational point of view, neuronal archetypes would thus constitute the words of a finite dictionary. Following this analogy, neurons would be the letters that form the syllables constituting words. Some words can be coupled, thus acquiring a more or less different meaning, and all the words can be concatenated to build (meaningful) sentences.

As an example of a well-known archetype, locomotive motion and other rhythmic behaviors are controlled by specific neuronal circuits called Central Generator Patterns (CPG) [5]. These CPGs have the capacity to generate oscillatory activities, at various regimes (under various different conditions), thanks to some specific properties at the circuit level.

It is relevant to investigate the dynamic behavior of all the possible archetypes of 2, 3 or more neurons, up to considering archetypes of archetypes. The aim is to see whether the properties of these archetypes of archetypes simply are an addition of the individual constituent archetypes properties or something more. Since it would be extremely difficult to prove the properties of archetypes expected from the biological theory through real biological experiments, we exploit formal methods, and this is our originality. Indeed, electrophysiological techniques that allow recording of spiking neuronal activity in vivo, or ex vivo, are mostly blind regarding the precise underlying anatomical structure. In other words, when one records the activity of some neurons in real tissues, one can at best determine through statistical methods that two or more neurons are functionally, but not anatomically, connected [6]. Nevertheless, this anatomical information is potentially accessible through in vitro recordings of neuronal cultures with Multiple Electrode Array techniques, for instance. However, the neuronal circuits formed in this kind of preparations are anarchic and thus useless for the current problem. A last possibility is to guide the connections of a few neurons during the circuit growth, thanks to some sorts of channels [7]. However again, the morphology of individual neurons in these kinds of preparations is far from being normal. That is why we promote the fact that formal methods can advantageously participate and help in answering certain questions related to Theoretical Neurosciences. In this paper, we report and compare the first attempts in the literature to apply *formal methods* of computer science to model and verify the temporal properties of fundamental neuronal archetypes in terms of neuronal information processing. We focus on the behavior of different basic archetypes and, for each one of them, we propose one or two representative properties that have been identified after extensive discussions with neurophysiologists [8, 9, 10]. From an electronic perspective, we consider archetypes as biologically inspired logical operators, which are easily adjustable by playing with very few parameters.

To model neuronal archetypes, we focus on Boolean Spiking Neural Networks where the neurons' electrical properties are described via the *leaky integrate and fire* (LI&F) model [11]. Notice that discrete modeling is well suited to this kind of model because neuronal activity, as with any recorded physical event, is only known through discrete recording (the recording sampling rate is usually set at a significantly higher resolution than the one of the recorded system, so that there is no loss of information). We describe neuronal archetypes as weighted directed graphs whose nodes represent neurons and whose edges stand for synaptic connections. At each time unit, all the neurons compute their membrane potential, accounting not only for the current input signals, but also for the ones received along a given temporal window. Each neuron can emit a spike when a given threshold is exceeded. This kind of modeling is more sophisticated than the one proposed by McCulloch and Pitts in [12], where the behavior of a neural network is expressed in terms of propositional logic and the present activity of each neuron does not depend on past events.

Our first approach is inspired by *formal verification*, which was initially introduced to prove that a piece of software or hardware is free of errors [13] with respect to a given model. The system at issue is generally modeled as a transition graph where each node represents a state of the system and

each edge stands for a transition from a source to a destination state. Model checkers are often used to verify that specific properties of the system hold at particular states. The field of systems biology is a more recent application area for formal verification, and such techniques have turned out to be very useful so far in this domain [14]. A variety of biological systems can be modeled as graphs whose nodes represent the different possible configurations of the system and whose edges encode meaningful configuration changes. It is then possible to define and prove properties concerning the temporal evolution of the biological species involved in the system [15, 16]. This often allows deep insight into the biological system at issue, in particular concerning the biological transitions governing it, and the reactions the system will have when confronted with external factors such as disease, medicine, and environmental changes [17, 18].

Our model checking approach was introduced by a subset of the authors of this paper [8, 9] and is based on the modeling of neural networks using a *synchronous language* for the description of reactive systems (Lustre). Spiking neural networks can indeed be considered as reactive systems: their inputs are physiological signals coming from input synapses, and their outputs represent the signals emitted in reaction. This class of systems fits well with the synchronous approach based on the notion of logical time [19]: time is considered as a sequence of logical discrete instants. An instant is a point in time where external input events can be observed, along with the internal events that are a consequence of the latter. Inputs and resulting outputs all occur simultaneously. The synchronous paradigm is now well established relying on a rigorous semantics and on tools for simulation and verification.

Several synchronous languages respect this synchronous paradigm, and all of them have a similar expressivity. We choose Lustre [19], which defines operator networks interconnected with data flows and is particularly well suited to expressing neuronal networks. It is a data flow language offering two main advantages: (1) it is *functional* with no complex side effects, making it well adapted to formal verification and safe program transformation; (2) it is a *parallel* model, where any sequencing and synchronization depends on data dependencies. Moreover, the Lustre formalism is close to temporal logic and this allows the language to be used for both writing programs and expressing properties as observers [20]. An observer of a property is a program, taking as inputs the inputs/outputs of the model under verification, and deciding at each instant whether the property is violated or not. The tools automatically checking the property satisfaction/violation are called *model checkers*. There exist several model checkers for Lustre that are well suited to our purpose: *Lesar* [21], *Nbac* [22], *Luke* [23], *Rantanplan* [24] and *kind2* [25, 26]. After comparing all these model checkers [8], the most powerful one turned out to be kind2, which relies on SMT (Satisfiabitily Modulo Theories) based k-induction and combines several resolution engines.

Nowadays model checking techniques are widely applied by industrial companies to develop safe critical systems ever since some important critical software failures have had harmful consequences. For instance, in the domain of cryptology, G. Lowe used the FDR model checker (based on CSP) to show that the well known Needham-Schroeder Public-Key Protocol can be attacked. This error had not been discovered for seventeen years despite numerous tests [27]. Similarly, mis-specifications (forgotten cases) have been automatically identified in a protocol for a contactless smart card, thanks to the symbolic handling of numerical constraints by Linear Decision Diagrams [28]. This method checks all states and verifies that the conjunction of transitions constraints always cover the complete space. On the other hand, formal methods and particularly model checking have been used in the development of the control system for the Maeslant Kering. The Maeslant Kering is a movable barrier which protects Rotterdam from flooding while, at almost the same time, not restricting shipping traffic to the port of Rotterdam [29]. In these two examples, the use of verification tools was the only way to ensure that safety properties are respected. Furthermore, since two decades, NASA has included model checking techniques in the verification and validation of advanced software, in the context of the generation of space shuttle [30]. Since model checking techniques explore models that are usually huge, issues of scaling up have been encountered. First symbolic model checking has been introduced to allow a concise representation of models and properties. For instance, it has been applied to verify sequential circuit the model of which has $5\times10\hat{}120$ states [31]. Moreover, in the last decades abstraction and SAT-solver methods have been introduced to allow large model exploration. In [32],

Clark and all show that abstraction and model checking allow to validate a program representing a pipelined ALU circuit with over 10^1300 states.

Complexity of temporal logic model checking grows exponentially when systems are composed of many parallel processes. To address this problem, Clarke et al. [33] have introduced a method to perform "compositional model-checking". The goal is to check properties of the components of a parallel composition and then deduce global properties from these local properties. Indeed, local properties are generally not preserved at the global level. Clarke et al. defined a general framework that uses an additional interface for each component that models its environment; then properties that hold for the composition of a component and its interface are preserved at the global level. In our approach, this technique applies since global neuronal networks are compositions of basic neuronal archetypes, defined themselves from behavior of single neurons.

Our second approach, which was also introduced by a subset of the authors of this paper [10], uses *theorem proving*. In general, theorem proving involves reasoning in a formal mathematical logic. Tools in this category generally implement the basic reasoning steps of a particular logic, such as a predicate logic or a higher-order logic, and include facilities for automating reasoning in these logics. In the broader context of formal methods, both model checking and theorem proving are on the more formal end of the spectrum, which means that they provide higher degrees of assurance than other formal methods, but at a greater cost in terms of the amount of effort required to apply them. Within the particular context of the methods we apply in our work, model checking provides more automation and requires less effort to apply than theorem proving, while theorem proving provides more flexibility in defining models and expressing more general properties about them. For example, using the full expressive power of a logic often leads to more direct and concise statements of properties, but requires more human interaction to guide the proofs.

Even within theorem proving, there is a large spectrum of automated and interactive theorem provers (see for example [34, 35]); less expressive logics are generally easier to automate, while more expressive logics provide more flexibility in defining models and proving properties about them. Fully automated tools do not scale as well as interactive tools with regard to the size and level of difficulty of the problem; on the other hand, interactive tools require more time and effort from the user. Modern interactive provers integrate a variety of automated functionality to help with easier and smaller subcases of proofs, thus increasing scalability. In our work, we use the Coq Proof Assistant, an interactive theorem prover that implements a highly expressive logic called the Calculus of Inductive Constructions [36]. It is a widely used system that won the prestigious ACM Software System Award in 2013. Other theorem provers in this category include Nuprl [37], the PVS Specification and Verification System [38], and Isabelle/HOL [39].

In recent decades, interactive theorem provers have been applied to a variety of large case studies in two distinct domains: verification of programs and systems, and formalization of large mathematical proofs. A prominent example in the first category is the CompCert Project [40], which includes a fully verified compiler for a large subset of C. The formal proof in Coq guarantees that the compiler introduces no bugs; a compiled program behaves exactly as specified by the original source program, according to the semantics of the C language. The CompCert compiler is only slightly slower than other well-known C compilers, which are known to have many bugs [41]. Another example in this category is the verification of the sel4 operating system microkernel in Isabelle/HOL [42]. In the domain of formal proofs in mathematics, theorem provers have been used to check a variety of handwritten proofs, often finding mistakes or omitted cases. Two large examples of fully formalized mathematical results include the Feit-Thompson theorem in algebraic group theory in Coq [43] and the Kepler conjecture in the HOL Light and Isabelle/HOL theorem provers [44]. The former took over 6 years to complete and involved formalizing several textbooks of background material in the process. Our own work falls mainly into the first category. Proofs of correctness of computer systems and (in our case) biological systems involve many technical details that are easy to get wrong; formal proof helps to automate the repetitive cases as well as guarantee that no cases are omitted.

Our development does not depend on advanced features of Coq like dependent types or the hierarchy of universes, and thus while we take into account everything about the model and properties, our

model is not difficult to understand, even for readers with a basic knowledge of theorem proving, and could likely be translated fairly easily to other interactive theorem provers such as those mentioned above. Some of Coq's features that were most useful in our work include its general facilities for defining datatypes, which we used to directly model neurons and archetypes, its powerful mechanisms for carrying out proofs by structural induction and case analysis, and its standard libraries that helped in reasoning about rational numbers and functions on them. One of the main advantages of using Coq for our purposes is the generality of its proofs; Coq can be used to prove properties about arbitrary values of parameters, such as any length of time, any input sequence, or any number of neurons. Of course, in any verification effort, whether model checking or theorem proving, the guarantees provided rely on the correctness of any assumptions made; in our setting, this means that we must do our best to correctly model neurons and archetypes.

The main contributions of this paper are:

- We define a Lustre model for a leaky integrate and fire neuron, six fundamental archetypes (simple series, series with multiple outputs, parallel composition, negative loop, inhibition of a behavior, and contralateral inhibition), and some representative couplings of these archetypes (simple series within negative loop, concatenation of simple series and negative loop, and series within contralateral inhibition). We also encode a pattern generator which generates periodic inputs and consider the concatenation of pattern generator and negative loop, and the concatenation of pattern generator and inhibition.

- We formalize the following properties in Lustre: delayer or filter in a single-input neuron, $n$-delayer or $n$-delayer/filter in a simple series, exclusive temporal activation in a series with multiples outputs, parallel composition of $n$ filters, oscillation in a negative loop, fixed point inhibition, winner takes all in a contralateral inhibition, oscillation period extension in a simple series within a negative loop, oscillation delay in concatenation of a simple series and a negative loop, and winner takes all delay in a series within a contralateral inhibition. All the properties have been verified thanks to the model checker Kind2 (for suitable parameter values).

- We define a Coq model for a leaky integrate and fire neuron and an abstract Coq archetype model using Coq's parameterized records.

- We formalize a variety of properties in Coq. The ones we focus on in this paper include the delayer effect for a single-input neuron, the delayer effect for a simple series, and the spike decreasing property for a single-input neuron. In these properties, there is no bound on the number of input values over time or the number of neurons in a simple series; the proofs cover all possible lengths of input values and series sizes. The accompanying code also includes proofs for some other properties such as properties about inhibition behavior and the filter effect, which have longer proofs and are not discussed in this paper.

- We complete a machine-checked proof for each property expressed in Coq. As discussed, formal proofs are guaranteed to cover all possible cases; no details are overlooked as is often the case in mathematical "on-paper" proofs.

- We carry out a rigorous comparison between the use of model checkers and theorem provers in the context of neuronal network verification.

Observe that the neuron properties we prove are intrinsic properties of the proposed model, that is, the neuron behavior expressed in our properties is the expected behavior of LI&F neurons. Other classes of properties, such as the ones dealing with inter-spike memory, do not hold in our modelling framework because LI&F neurons are not influenced by the history preceding the last spike. Other models, such as the Hodgkin–Huxley model [45], are known for being able to reproduce more behaviors. However, their computational complexity is high and they are not amenable for formal verification.

To the best of our knowledge, the two approaches reported in this paper (the model-checking oriented approach and the theorem proving approach) are the only approaches to the formal verification

of neuronal archetypes. The paper is organized as follows. In Section 2, we present the state of the art concerning neural network modeling and, more generally, modeling and formal verification of biological systems. In Section 3, we present a discrete version of the LI&F model. Section 4 is devoted to the description of the neuronal archetypes we take into consideration. In Section 5, we introduced background material on Lustre and Coq needed to understand the sections that follow. In Section 6, we present the details of the model checking approach. In the first subsection, we show archetype behaviors (encoded in Lustre) and in the second subsection, we tackle the next logical step and deal with some archetype couplings. In Section 7, we present the details of the theorem proving approach. In the first subsection, we present the Coq model for neural networks, which includes definitions of neurons, operations on them, and combining them into archetypes. In the second subsection, we present and discuss the Coq specification and proof of three representative properties (the first two properties are intentionally two of the properties of Section 6.2). Finally, in Section 8 we summarize and compare the two proposed approaches on the formal verification of neuronal archetypes and give some future research directions. This paper has the value of rigorously comparing the approaches proposed in the conference papers [8, 9, 10]. Furthermore, it extends these papers by introducing new archetype definitions and properties. The first author of this paper belongs to the intersection of the author lists of the cited papers.

## 2 Related Work

In the last decades, there has been much promising research in the field of formal modelling and verification of biological systems. As far as the modelling of biological systems is concerned, in the literature we can find both qualitative and quantitative approaches. To express the qualitative nature of dynamics, the most used formalisms are Thomas' discrete models [46], Petri nets [47], p-calculus [48], bio-ambients [49], and reaction rules [50]. To capture the dynamics from a quantitative point of view, ordinary or stochastic differential equations are often used. More recent approaches include hybrid Petri nets [51] and hybrid automata [52], stochastic p-calculus [53], and rule-based languages with continuous/stochastic dynamics such as Kappa [54]. Relevant properties concerning the obtained models are then often expressed using a formalism called temporal logic and verified thanks to model checkers such as NuSMV [55] or PRISM [56].

Concerning the theorem proving approach, in [57] the authors propose the use of modal linear logic as a unified framework to encode both biological systems and temporal properties of their dynamic behavior. They focus on a model of the P53/Mdm2 DNA-damage repair mechanism and they prove some desired properties using theorem proving techniques. In [58], the Coq Proof Assistant is exploited to prove two theorems linking the topology and the dynamics of gene regulatory networks. In [59], the authors advocate the use of higher-order logic to formalize reaction kinetics and exploit the HOL Light theorem prover to verify some reaction-based models of biological networks. Finally, the Porgy system is introduced in [60]. It is a visual environment which allows modeling of biochemical systems as rule-based models. Rewriting strategies are used to choose the rules to be applied.

As far as neuronal networks are concerned, their modeling is classified into three generations in the literature [61, 62]. First generation models, based on McCulloch-Pitts neurons [12] as computational units, handle discrete inputs and outputs. Their computational units consist of a set of logic gates with a threshold activation function. Second generation models, whose most representative example is the multi-layer perceptron [63], exploit real valued activation functions. These networks, whose real-valued outputs represent neuron firing rates, are widely used in the domain of artificial intelligence. Third generation networks, also called *spiking neural networks* [62], are characterized by the relevance of time aspects. Precise spike firing times are taken into account. Furthermore, they consider not only current input spikes but also past ones (temporal summation). In [64], spiking neural networks are classified with respect to their biophysical plausibility, that is, the number of behaviors (i.e., typical responses to an input pattern) they can reproduce. Among these models, the Hodgkin-Huxley model [45] is the one able to reproduce most behaviors. However, its simulation process is very expensive even for a few neurons and for a small amount of time. In this work, we choose to use the leaky integrate

and fire (LI&F) model [65], a computationally efficient approximation of a single-compartment model, which proves to be amenable to formal verification.

In addition to the works reviewed in this paper [8, 9, 10], there are a few attempts at giving formal models for spiking neural networks in the literature. In [66], a mapping of spiking neural P systems into timed automata is proposed. In that work, the dynamics of neurons are expressed in terms of evolution rules and durations are given in terms of the number of rules applied. Timed automata are also exploited in [67] to model LI&F networks. This modeling is substantially different from the one proposed in [66] because an explicit notion of duration of activities is given. Such a model is formally validated against some crucial properties defined as temporal logic formulas and is then exploited to find an assignment for the synaptic weights of neural networks so that they can reproduce a given behavior.

## 3 Discrete Leaky Integrate and Fire Model

In this section, we introduce a discrete (Boolean) version of LI&F modeling. We first present the basic biological knowledge associated to the modeled phenomena and then we detail the adopted model.

When a neuron receives a signal at one of its synaptic connections, it produces an excitatory or an inhibitory *post-synaptic potential* (PSP) caused by the opening of selective ion channels according to the post-synaptic receptor nature. An inflow of cations in the cell leads to an activation; an inflow of anions in the cell corresponds to an inhibition. This local ions flow modify the membrane potential either through a depolarization (excitation) or a hyperpolarization (inhibition). Such variations of the membrane potential are progressively transmitted to the rest of the cell. The potential difference is called *membrane potential*. In general, several to many excitations are necessary for the membrane potential of the post-synaptic neuron to exceed its *depolarization threshold*, and thus to emit an *action potential* at its axon hillock to transmit the signal to other neurons.

Two phenomena allow the cell to exceed its depolarization threshold: the *spatial summation* and the *temporal summation* [68]. Spatial summation allows to sum the PSPs produced at different areas of the membrane. Temporal summation allows to sum the PSPs produced during a finite time window. This summation can be done thanks to a property of the membrane that behaves like a capacitor and can locally store some electrical loads (*capacitive property*).

The neuron membrane, due to the presence of leakage channels, is not a perfect conductor and capacitor and can be compared to a resistor inside an electrical circuit. Thus, the range of the PSPs decreases with time and space (*resistivity* of the membrane).

A LI&F neuronal network is represented with a weighted directed graph where each node stands for a neuron soma and each edge stands for a synaptic connection between two neurons. The associated weight for each edge is an indicator of the weight of the connection on the receiving neuron: a positive (resp. negative) weight is an activation (resp. inhibition).

The depolarization threshold of each neuron is modeled via the *firing threshold* $\tau$, which is a numerical value that the neuron membrane potential $p$ shall exceed at a given time $t$ to emit an action potential, or *spike*, at the time $t + 1$.

The membrane resistivity is symbolized with a numerical coefficient called the *leak factor* $r$, which allows to decrease the range of a PSP over time.

Spatial summation is implicitly taken into account. In our model, a neuron $u$ is connected to another neuron $v$ via a single synaptic connection of weight $w_{uv}$. This connection represents the entirety of the shared connections between $u$ and $v$. Spatial summation is also more explicitly taken into account with the fact that, at each instant, the neuron sums each signal received from each input neuron. The temporal summation is done through a sliding integration window of length $\sigma$ for each neuron to sum all PSPs. Older PSPs are decreased by the leak factor $r$. This way, the biological properties of the neuron are respected and the computational load remains limited. This allows us to obtain finite state sets, and thus to easily apply model checking techniques.

More formally, the following definition can be given:

**Definition 1** ***Boolean Spiking Integrate and Fire Neural Network.*** A spiking Boolean integrate and fire neural network *is a tuple* $(V, E, w)$*, where:*

- $V$ *are Boolean spiking integrate and fire neurons,*
- $E \subseteq V \times V$ *are synapses,*
- $w : E \to \mathbb{Q} \cap [-1, 1]$ *is the synapse weight function associating to each synapse* $(u, v)$ *a weight* $w_{uv}$*.*

A spiking Boolean integrate and fire neuron *is a tuple* $(\tau, r, p, y)$*, where:*

- $\tau \in \mathbb{Q}^+$ *is the* firing threshold*,*
- $r \in \mathbb{Q} \cap [0, 1]$ *is the* leak factor*,*
- $p : \mathbb{N} \to \mathbb{Q}$ *is the [membrane]* potential *function defined as*

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

  *where* $p(0) = 0$*,* $m$ *is the number of inputs of the neuron,* $w_i$ *is the weight of the synapse connecting the* $i^{th}$ *input neuron to the current neuron, and* $x_i(t) \in \{0, 1\}$ *is the signal received at the time* $t$ *by the neuron through its* $i^{th}$ *input synapse (observe that, after the potential exceeds its threshold, it is reset to* $0$*),*

- $y : \mathbb{N} \to \{0, 1\}$ *is the neuron output function, defined as*

$$y(t) = \begin{cases} 1 & if\, p(t) \geqslant \tau \\ 0 & otherwise. \end{cases}$$

  *(for the Lustre implementation, we set* $y = 1$ *if* $p(t-1) \geqslant \tau$ *in order to prevent neurons from receiving and emitting signals at the same time unit.)*

The development of the recursive equation for the membrane potential function and the introduction of a sliding time window of length $\sigma$ lead to the following equation for $p$ (when $p(t-1) < \tau$): $p(t) = \sum_{e=0}^{\sigma} r^e \sum_{i=1}^{m} w_i \cdot x_i(t-e)$, where $e$ represents the time elapsed until the current one. The sliding window is useful for the Lustre implementation.

# 4 Case Studies

In this paper we consider six case studies, which are the basic archetypes graphically depicted in Figure 1. These archetypes can be coupled to potentially constitute a bigger one.

- **Simple series** is a sequence of neurons where each element of the chain receives as input the output of the preceding one.
- **Series with multiple outputs** is a series where, at each time unit, we are interested in knowing the outputs of all the neurons (i.e., all the neurons are considered as output ones).
- **Parallel composition** is a set of neurons receiving as input the output of a given neuron.
- **Negative loop** is a loop consisting of two neurons: the first neuron activates the second one while the latter inhibits the former.
- **Inhibition of a behavior** consists of two neurons, the first one inhibiting the second one.
- **Contralateral inhibition** consists of two or more neurons, each one inhibiting the other ones.

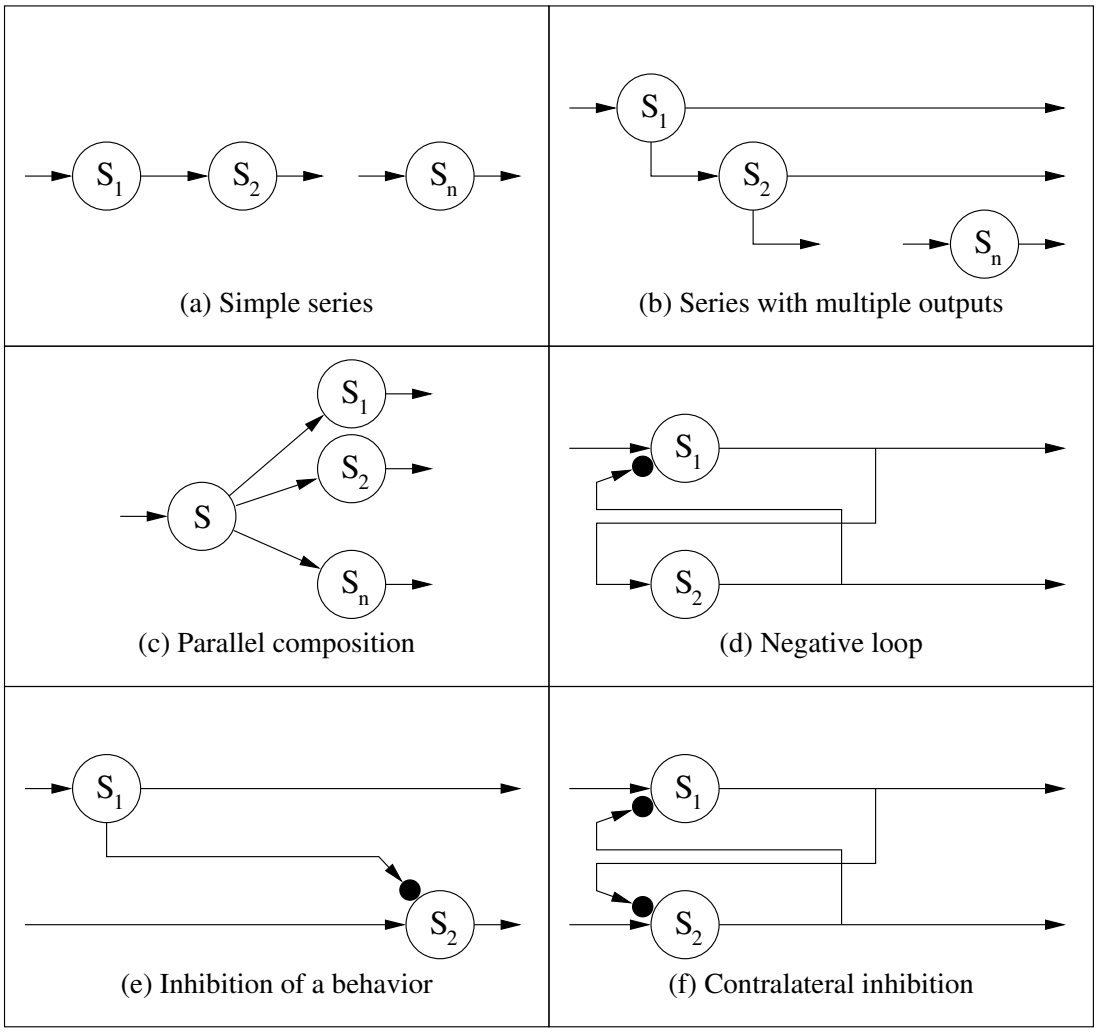


Figure 1: The basic neuronal archetypes.

# 5 Background

In Section 5.1, we present the Lustre language used in the model checking approach, and in Section 5.2, we present the Coq Proof Assistant used in the theorem proving approach.

## 5.1 The Synchronous Language Lustre and Model Checking

As explained in Section 1, we adopt the synchronous language Lustre [19] to describe neuronal behaviors with a declarative modeling approach. Such a programming language is based on the *synchronicity hypothesis*, which is characterized by the concept of *logical time*. Lustre considers *time* as a sorted discrete *flow* of signals. At each time unit (*clock* value), systems react to inputs and generate outputs at the same instant. A Lustre code displays time dependencies between signals and explains them thanks to a set of equations. This mechanism allows to express time as an infinite sequence of (natural) values. A Lustre program behaves as a cyclic system: all the present variables take their $n$th value at the $n$th execution step.

The basic structure of a Lustre program is the *node*. In a Lustre node, equations, expressions, and assertions are used to express output variable sequences of values from input variable sequences. Variables are typed: types can be either basic (Boolean, integer, real), or complex (structure or vector). Complex types are defined by the user. Usual operators over basic types exist: `+, -, ...`; `and, or, not`; `if then else`. These data operators only work with variables sharing the same clock. The result

shares the same clock too. Furthermore, Lustre disposes of two main temporal operators to handle logical time represented by clocks. These operators are `pre` and $\rightarrow$:

- `pre` (for previous) reacts as a memory: if $(e_1, e_2, ....., e_n, \dots)$ is the flow `E`, `pre(E)` is the flow $(nil, e_1, e_2, ..., e_n, \dots)$, where $nil$ is the undefined value denoting uninitialized memory.
- $\rightarrow$ (meaning "followed by") is used with the `pre` operator and prevents from having uninitialized memory: let `E` $= (e_1, e_2, \dots, e_n, \dots)$ and `F` $= (f_1, f_2, \dots, f_n, \dots)$ be two flows, then `E`$\rightarrow$`F` is the expression
  $(e_1, f_2, \dots, f_n, \dots)$.

Equations define output variables in nodes. As an example, to generate the flow of odd and even numbers we can specify the following Lustre node:

```
node odd_even (freeze:bool) returns (odd, even:int)
let
   odd = 0 -> if freeze then pre(odd)
              else pre(odd)+2;
   even = 1 -> if freeze then pre(even)
               else pre(even)+2;
tel
```

In the code above, we can see that the variable `freeze` is able to change the behavior of the node: when it is true, the output variables are frozen; when it is false, the `odd` and `even` variables increase as expected.

The corresponding timetable is given in Figure 2.

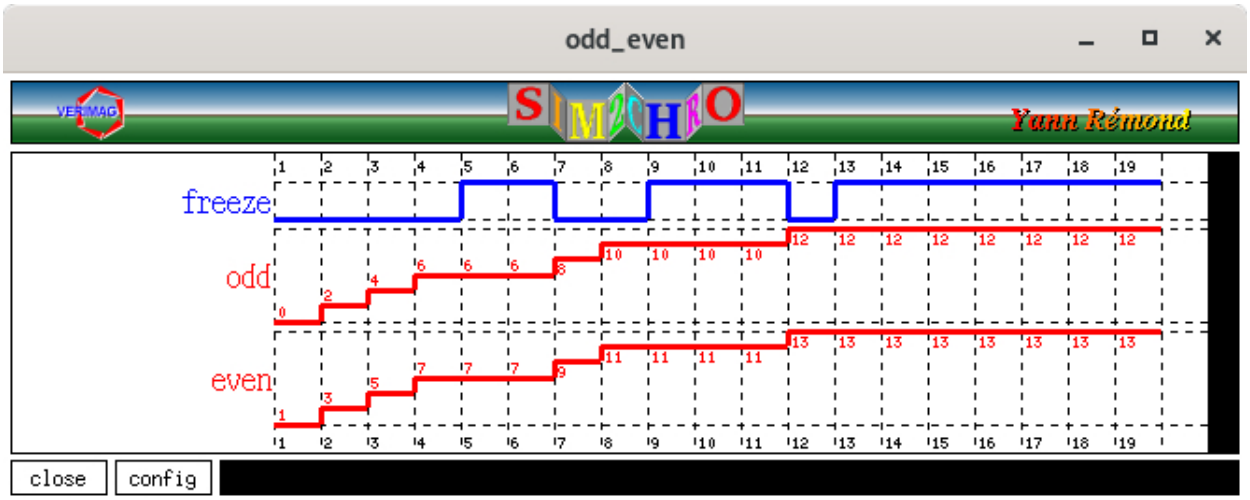


Figure 2: Timetable corresponding to the execution of the node `odd_even`.

Moreover, assertions can be exploited to force variable values. Assertions allow to take the environment into account by making assumptions which consist in boolean expressions. These expressions must always be true. For instance, the assertion:
`assert (true` $\rightarrow$ `(not x) or (pre(not x)))` says that in the value sequence of $x$, no two consecutive `true` exist.

Lustre is a unifying framework: on one side, it allows to model reactive systems, on the other side it gives the possibility to express some temporal properties concerning the modeled systems. These properties are written as Lustre nodes called *observers* [20], which verify the outputs of the program under verification for each possible instant, and return `true` if the encoded behavior is satisfied, `false` otherwise. The tool kind2 [25, 26] allows to make automatic verifications. We have chosen it for its

compatibility with the Lustre syntax and for its efficiency compared to other model checkers such as Lesar [21] and Nbac [22]. If the given property is not true for all possible input variable values, then kind2 gives a counterexample, which consists in an execution trace leading to the violation of the property.

### 5.2 The Coq Proof Assistant

In this section, we introduce the main Coq features we exploit for neural network modeling. Additional information on Coq can be found in [36, 69]. The full implementation of our model along with the properties and their proofs described in this section, Section 7, and the appendix are available at

`http://www.site.uottawa.ca/~afelty/coq-archetypes/`.

Coq is a formal proof management system that implements the Calculus of Inductive Constructions [70], which is an expressive higher-order logic [71]. Such a system allows users to formalize and prove properties in this logic. Expressions in the logic include a functional programming language. Such a language is typed (every Coq expression has a type). For instance, `X:nat` says that variable `X` takes its value in the domain of natural numbers. The types we employ in our model include `nat`, `Q`, `bool`, and `list` which denote natural numbers, rational numbers, booleans, and lists of elements respectively. These types are available in Coq's standard libraries. All the elements of a list must have the same type. For example, `L:list nat` expresses that `L` is a list of natural numbers. An empty list is denoted by `[]` or `nil` in Coq. Functions are the basic components of functional programming languages. The general form of a Coq function is given below.

`Definition`/`Fixpoint` Function_Name
(Input$_1$: Type of Input$_1$) ...
(Input$_n$: Type of Input$_n$): Output Type :=
  Body of the function.

The Coq keywords `Definition` and `Fixpoint` are used to define non-recursive and recursive functions, respectively. These keywords are followed by the function name. After the function name, there are the input arguments and their corresponding types. Inputs having the same type can be grouped. For instance, `(X Y Z: Q)` states all variables `X`, `Y`, and `Z` are rational numbers. Inputs are followed by a colon, which is followed by the output type of the function. Finally, there is a Coq expression representing the body of the function, followed by a dot.

In Coq, pattern matching is exploited to perform case analysis. This useful feature is used, for instance, in recursive functions, for discriminating between base cases and recursive cases. For example, it is employed to distinguish between empty and nonempty lists. A non-empty list consists of the first element of the list (the *head*), followed by a double colon, followed by the rest of the list (the *tail*). The tail of a list itself is a list and its elements have the same type as the head element. For instance, let $L$ be the list `(5::2::9::nil)` containing three natural numbers. In Coq, the list $L$ can also be written as `[5;2;9]`, where the head is `5` and the tail is `[2;9]`. Thus, non-empty lists in Coq often follow the general pattern `(h::t)`. In addition, there are two functions in Coq library called `hd` and `tl` that return the head and the tail of a list, respectively. For example, `(hd d l)` returns the head of the list `l`. Here, `d` is a default value returned if `l` is an empty list and thus does not have a head. Also, `(tl l)` returns the tail of the list `l` and returns `nil` if there is no tail.

Another Coq data type is natural numbers. A natural number is either `0` or the successor of another natural number, written `(S n)`, where `n` is a natural number. For instance, `1` is represented as `(S 0)`, `2` as `(S (S 0))`, etc. In the code below, some patterns for lists and natural numbers are shown using Coq's `match`...`with`...`end` pattern matching construct.

```
match X with
| 0 ⇒ calculate something when X = 0
| S n ⇒ calculate something when X is successor of n
end
```

```
match L with
| [] ⇒ calculate something when L is an empty list
| h::t ⇒ calculate something when L has head h
          followed by tail t
end
```

In addition to the data types that are defined in Coq libraries, new data types can be introduced. One way to do so is using Coq's facility for defining records. Records can have different fields with different types. For instance, we can define a record with three fields `Fieldnat`, `FieldQ`, and `ListField`, which have types natural number, rational number, and list of natural numbers, respectively. Fields in Coq records can also represent constraints on other fields. For instance, field `CR` in the code below states that field `Fieldnat` must be greater than 7. The Coq syntax for the definition of the full record is shown in the code below.

```
Record Sample_Record := MakeSample {
  Fieldnat: nat;
  FieldQ: Q;
  ListField: list nat;
  CR: Fieldnat > 7 }.

S: Sample_Record
```

A record is a type like any other type in Coq, and so variables can have the new record type. For example, variable `S` with type `Sample_Record` is an example. When a variable of this record type gets a value, all the constraints in the record have to be satisfied. For example, `Fieldnat` of `S` cannot be less than or equal to 7.

# 6 Model Checking Approach and Experiments

This section is devoted to our model checking approach to proving dynamic properties of neurons, archetypes, and their coupling. In Section 6.1, we report on the experiment setup. In Section 6.2, we present the encoding of neurons and archetypes and some temporal properties in Lustre. In Section 6.3, we discuss archetype coupling and corresponding properties in Lustre.

## 6.1 Experimental Setup

All the experiments of this part are performed on a personal computer with the following characteristics: Proc Intel Core i3-4005U CPU 1.70GHz*4, 3.8 GB Memory, operating system Ubuntu 16.04 LTS. We encode neuronal networks of maximal size equal to 20 (this size is reached for archetype coupling). The model checking computation times with Kind2 are at most 0.15 seconds for single neurons, 0.25 seconds for archetypes, 0.36 seconds for archetype coupling, and 53.55 seconds for compositions including the pattern generator. Maximum resident set size (the portion of memory occupied by a process that is held in main memory) for Kind2 is around 17 Mbytes for single neurons, 18.5 Mbytes for archetypes, 19 Mbytes for archetype coupling, and 51 Mbytes for compositions including the pattern generator.

## 6.2 Encoding Neuronal Archetypes and Temporal Properties in Lustre

Lustre allows easy modeling and encoding of neuron behaviors. An input matrix (`mem`) is used to record present and past received signals. For each instant, the leftmost column of this matrix stores the (weighted) current inputs and, for the other columns, values are defined as follows: (i) they are equal to 0 at the first instant (initialization) and (ii) for all the next instants, they are reset to 0 in case of spike emission at the previous instant, and they take the preceding time unit value of their own left column (for the corresponding row) otherwise. This process implements a sliding time window

and is encoded with the `pre` operator. Such an input matrix is multiplied, at each instant, by a vector of remaining coefficients (`rvector`) and, whenever the firing threshold is reached, a spike is emitted (at the next time unit). The code defining a Lustre neuron with one input data stream is given below. In the node `neuron`, `X` is the spike input stream and `w` is the weight of the input edge.

```
node neuron (X: bool; w:int) returns(Spike: bool);
var threshold, V: int;
    rvector: int^5;
    mem: int^5;
    localS: bool;
let
   threshold=105;
   rvector[0..4]=[10,5,3,2,1];
   V=mem[0]*rvector[0]+mem[1]*rvector[1]+
     mem[2]*rvector[2]+mem[3]*rvector[3]+
     mem[4]*rvector[4];
   localS=(V>=threshold);
   mem[0]=if X then w else 0;
   mem[1..4]=[0,0,0,0]->if pre(localS) then 0
                        else pre(mem[0..3]);
   Spike= false -> pre(localS);
   -- reaction at the next instant
tel
```

All the values are multiplied by ten in order to work in fixed point precision, and thus have better model checking performances. Notice that all the constants of this Lustre node can be given as parameters as follows:

```
node neuron (X: bool; w, threshold: int;
             rvector: int^5)
 returns(Spike: bool);
```

Lustre is a modular language and, thanks to this feature, archetypes can be directly encoded from basic neurons. As a first example, the Lustre code for a simple series composed of three neurons of type `neuron` is given in the code below.

```
node series3 (X:bool; w:int) returns(Spike:bool);
  var chain: bool^3;
  let
    chain[0]=neuron(X,w);
    chain[1..2]=neuron(chain[0..1],w);
    Spike=chain[2];
  tel
```

In the node `series3`, each position of the vector `chain` refers to a different neuron of the chain. As far as the first neuron is concerned, it is enough to call the node `neuron` with the input `X` of the series as first input. For the other neurons of the chain, their behavior is modeled by calling `neuron` with the output of the preceding neuron as first input. The output of the node is the output of the last neuron of the series.

As a second example, we give the encoding of the negative loop. As illustrated in Figure 1(d), let us suppose to have a two-input activator neuron and a one-input inhibitor neuron. The Lustre interface of the activator and inhibitor neurons, `neuron_act` and `neuron_inh`, are given in the code below. In `neuron_act`, `X1` (resp. `X2`) is the activating (resp. inhibiting) input and `w1` (resp. `w2`) the corresponding weight.

```
node neuron_act (X1,X2:bool; w1,w2:int)
 returns(Spike:bool);
node neuron_inh (X:bool; w1:int)
 returns(Spike:bool);
```

The Lustre code for the negative loop archetype is the following one.

```
node negative_loop (X:bool; w1,w2,w3:int)
 returns(Spike_act, Spike_inh:bool);
  var mem0,mem1:bool;
  let
    mem0=neuron_act(X,mem1,w1,w2);
    mem1=neuron_inh(mem0,w3);
    Spike_act=mem0;
    Spike_inb=mem1;
  tel
```

In the node `negative_loop`, the activator neuron takes as first input the input `X` of the archetype and as second input the output of the inhibitor neuron (`mem1`); it stocks the result in `mem0`. On the other hand, the inhibitor neuron takes as first input the output of the activator neuron (`mem0`) and stocks the result in `mem1`. The first (resp. second) output of this node is `mem0` (resp. `mem1`), which is the output of the activator (resp. inhibitor) neuron.

In the rest of this subsection, we focus on relevant properties of neurons and archetypes (we use 1 to denote `true` and 0 to denote `false`).

### 6.2.1 Single Neurons

Firstly, we concentrate on single neuron behaviors. Whatever the parameters of a neuron are (input synaptic weights, firing threshold, leak factor, length of the integration window), it can only behave in one of the two following ways: (i) the sum of the current entries (multiplied by their weights) allows to reach the threshold (and thus, for each time units when the neuron gets enough input signals, it emits a spike at the next instant), or (ii) several time units are required to overtake the threshold. More formally, given a neuron with a unique input, property 1 has been checked thanks to kind2:

**Property 1** ***[Delayer or filter in a single-input neuron.]*** *Given a neuron receiving an input flow on the alphabet* $\{0, 1\}$*, it can only express one of the two exclusive following behaviors:*

**Delayer effect.** *It emits a* 0 *followed by a flow identical to the input.*

**Filter effect.** *It emits at least two occurrences of* 0 *at the beginning and can never emit two consecutive occurrences of* 1*.*

From a biological point of view, in the first case (delayer) we can speak of instantaneous integrator and in the second case (filter) of long time integrator. In the case of a delayer, the neuron just emits a sequence identical to the input, with a delay of one time unit. In the case of a filter, the neuron only transmits one signal out of $x$ ($1/x$ filter). Note that if the neuron is not able to overtake its threshold (*wall* effect), we have a limit case of the filter effect. The Lustre observer of property 1, consisting of an exclusive conjunction between a delayer and a filter, is given in the code below.

```
node delayer (X: bool; w: int) returns(OK: bool);
  var
    SX, Out, S1, verif, preverif: bool;
  let
    Out = neuron(X, w);
    S1 = true -> Out;
    SX = false -> pre(X);
    verif  = true->if SX then S1 else false;
    preverif = true->pre(verif);
    OK = if preverif and verif then true else false;
  tel
```

In the node `delayer`, the output `OK` is true if the output of `neuron` follows its input of one time unit.

```
node filter (X: bool; w: int) returns(OK: bool);
  var
    Out: bool;
  let
    Out = neuron(X,w);
    OK =true -> if Out then not pre(Out)
                else not Out;
  tel
```

In the node `filter`, we check whether, if the output of the neuron `Out` is true, then it was false at the previous time unit.

```
node prop1 (X: bool; w: int) returns (OK: bool);
  var
    S1,S2, Verif: bool;
  let
    S1 = delayer(X, w);
    S2 = filter(X, w);
    Verif = S1 XOR S2;
    OK = confirm_property_2_ticks(X) or Verif;
  tel
```

Because of the initialization, the properties can only become true after two ticks. To avoid this problem, we integrate a node `confirm_property_2_ticks`, which is able to force the properties to true only for the first two ticks.

### 6.2.2 Simple Series (see Fig. 1(a))

A simple series of length $n$ behaves according to the neurons composing it. There are two cases: (i) the series contains $n$ delayers (and in this case it acts as a delayer of $n$ time units), or (ii) it contains at least one filter (it thus shows a $n$-delayer effect composed of a filter effect). More formally, the following property can be given:

**Property 2** *[**$n$-delayer or $n$-delayer/filter in a simple series.**] Given a series of length $n$ receiving an input flow on the alphabet $\{0, 1\}$, it can only express one of the two exclusive following behaviors:*

**$n$-delayer effect.** *It emits a sequence of $0$ of length $n$ followed by a flow identical to the input one.*

**$n$-delayer/filter effect.** *It emits a sequence of $0$ of length at least $n + 1$ and can never emit two consecutive $1$.*

A consequence of property 2 is that a simple series cannot constitute a permanent signal, e.g., if it receives an oscillatory signal as input, it is not able to emit a sequence of 1 as output. We can also point out that filter neurons do *not commute* in a simple series. For instance, if in a series of two neurons a 1/2 filter (that is, a neuron emitting a 1 every two instants when receiving a sequence of 1) precedes a 1/3 filter, the result is a 1/6 filter. If the two neurons are inverted, the result is a wall effect (no signals are emitted). In order to avoid wall effects, the most selective neurons should thus be the first ones.

#### 6.2.3 Series with Multiple Outputs (see Fig. 1(b))

In a series with multiple outputs, the emission of each neuron depends by the emissions of the preceding neurons. Furthermore, the following property is valid:

**Property 3** ***[Exclusive temporal activation in a series with multiples outputs.]*** *When a series of $n$ delayers with multiples outputs receives the output of a $1/n$ filter, only one neuron at a time overtakes its threshold (and thus emits a spike).*

As a consequence, in the configuration of Property 3 two neurons can not emit at the same time.

#### 6.2.4 Parallel Composition (see Fig. 1(c))

As far as the parallel composition is concerned, the number of spikes emitted in parallel at each instant is in between a given interval, whose upper bound is not necessarily the number of neurons in parallel (because, even if each single neuron has the capability to emit, the parallel neurons can be unsynchronized, that is, not able to emit simultaneously).

Moreover, the following relevant property holds:

**Property 4** ***[Parallel composition of $n$ filters.]*** *Given a parallel composition with a delayer connected to $n$ filters of different selectivity connected to the same delayer, it is possible to emit as output a sequence of* 1 *of length $k$, with $k \geq n$.*

To have a sequence of 1 of length $k$, the key idea is to take the parallel composition of $n$ filters of different selectivity $1/X_i$, where $i \in \{1, ., n\}$ and $X_1$,...,$X_n$ is the set of prime numbers in between 2 and $k$. Moreover, the parallel composition can be exploited to constitute a permanent signal from a not permanent one.

#### 6.2.5 Negative Loop (see Fig. 1(d))

If, without considering the inhibiting edge, the two neurons of a negative loop operate as delayers, then the two neurons show an oscillatory output behavior (provided that a permanent signal is injected in the archetype). More precisely, the following property is verified:

**Property 5** ***[Oscillation in a negative loop.]*** *Given a negative loop composed of two delayers, when a sequence of* 1 *is given as input, the activator neuron oscillates with a pattern of the form* 1100 *(and the inhibitor expresses the same behavior delayed of one time unit).*

The Lustre observer for the activator neuron of property 5 is given in the code below.

```
node oscil(X:bool; w1,w2,w3:int) returns(OK: bool);
  var S1,S2,Out:bool;
  let Out1,Out2=negative_loop(X,w1,w2,w3);
      S1=true->pre(Out);
      S2=true->pre(S1);
      S=if Out then not S2 else S2;
  tel
```

Let `negative_loop` be the Lustre node encoding the negative loop archetype and let `Out1` be the output of the activator neuron, as explained at the beginning of the section. In the node `oscil` we check that (i) if `Out1` is true, then it was false two time units ago and (ii) if `Out1` is false then it was true two time units ago.

The Lustre observer for the inhibitor neuron of property 5 is given in the code below.

```
node follow (X:bool; w1,w2,w3:int) returns(OK: bool);
var
  Out1,Out2,preOut1:bool;
let
  Out1,Out2=negative_loop(X,w1,w2,w3);
  preOut1=false->pre(Out1);
  OK=(Out2=preOut1);
tel
```

In the node `follow` we check that `Out2`, the output of the inhibitor neuron, is the same as `Out1`, the output of the inhibitor neuron, with a delay of one time unit.

#### 6.2.6 Inhibition of a Behavior (see Fig. 1(e))

For a large class of neuron parameters (that is, firing threshold, leak factor, and length of the integration window), property 6 holds:

**Property 6** ***[Fixed point inhibition.]*** *Given an inhibition archetype, if a sequence of* 1 *is given as input, at a certain time the inhibited neuron can only emit* 0 *values.*

#### 6.2.7 Contralateral Inhibition (see Fig. 1(f))

It has been shown that there is a set of neuron parameters for which the following behavior can be observed.

**Property 7** ***[Winner takes all in a contralateral inhibition.]*** *Given a contralateral inhibition archetype with two or more neurons, if a sequence of* 1 *is given as input, starting from a given time one neuron is activated and the other ones are inhibited.*

The intuition is that the most excited neuron becomes the most inhibitory one, and even if it is itself inhibited, it necessarily is less inhibited than its neighbors.

Some additional properties related to the different archetypes can be found in [72]. We are currently studying some supplementary archetypes such as the positive loop, where a first neuron activates a second neuron, which in turn activates the former one. While the negative loop is very frequent and allows to regulate systems, the negative loop is less frequent. On one hand, it allows to extend signals in time, on the other hand it can provoke a chain reaction where the system activates itself up to reaching its maximum, without having the capability of leaving it. It is interesting to find the conditions under which this phenomenon can be observed.

### 6.3 Archetype Coupling in Lustre

Two archetypes can be coupled in the following ways: (i) by connecting the output(s) of one archetype to the input(s) of the other one, that is, by making a concatenation, (ii) by nesting one archetype within the other one. In this subsection we introduce some representative couplings among the ones we treated. In the following figures, these acronyms are used: $A$ for activator, $I$ for inhibitor, $D$ for delayer, $F$ for filter, $G$ for pattern generator, $S$ for series, and $C$ for collector.

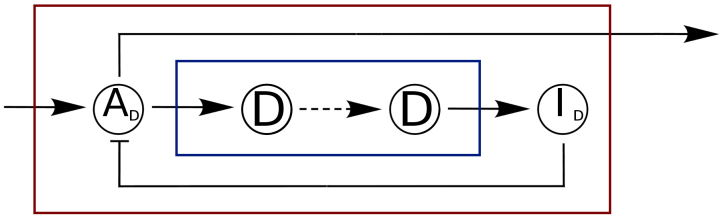


Figure 3: Series of $n$ delayers nested between the activator and the inhibitor of a negative loop.

### 6.3.1 Simple Series within Negative Loop

We start considering a series of $n$ delayers nested between the activator and the inhibitor of a negative loop, as illustrated in Figure 3. We remind that, when a sequence of 1 is injected in the negative loop archetype, an oscillating output of the form 1100 is produced (see Property 5). The addition of the series involves an augmentation of the oscillation period, which passes from 2 to $n$. More precisely, the following property holds:

**Property 8** ***[Oscillation period extension in a simple series within a negative loop.]*** *Given a simple series of delayers of length $n$ within a negative loop, if a sequence of* 1 *is given as input, the output of the activator is of the form :* $0(1^{n+2}0^{n+2})^{\omega}$ *(we recall that $x^y$ denotes the repetition of $x$ for $y$ time units and $x^{\omega}$ denotes the infinite repetition of $x$.)*

### 6.3.2 Concatenation of Simple Series and Negative Loop

We consider now a series of $n$ delayers preceding a negative loop, as illustrated in Figure 4. In this

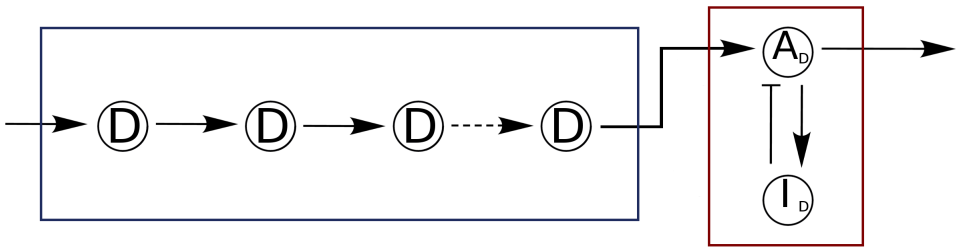


Figure 4: Series of $n$ delayers connected to the activator of a negative loop.

case, the oscillation period is not modified but the beginning of the oscillation is delayed. More in detail:

**Property 9** ***[Oscillation delay in concatenation of a simple series and a negative loop.]*** *Given a simple series of delayers of length $n$ connected to the activator of a negative loop, if a sequence of* 1 *is given as input, the output of the activator is of the form :* $0^n(1100)^{\omega}$.

As a next step, it is important to identify all the input patterns of the negative loop producing an oscillatory trend output. Since the simple series is only able to transmit or filter signals (see Property 2), in the next subsection we present a neuron combination which is able to produce all the patterns of a fixed length on the alphabet $\{0, 1\}$.

### 6.3.3 Concatenation of Periodic Pattern Generator and Negative Loop

Let us consider the system which is graphically depicted in Figure 5. As input of the system, a sequence of 1 is received by a $1/n$ filter connected to a series of $n$ delayers with multiple outputs, which is then connected to a collector delayer neuron. One can choose to activate or deactivate each one of the edges linking the neurons of the series to the collector. We could prove that this system can generate all the patterns of length $n$ on the alphabet $\{0, 1\}$. To get the different patterns, the intuition is to play with the activation/deactivation combinations of the edges connecting the neurons of the series to the collector neuron.

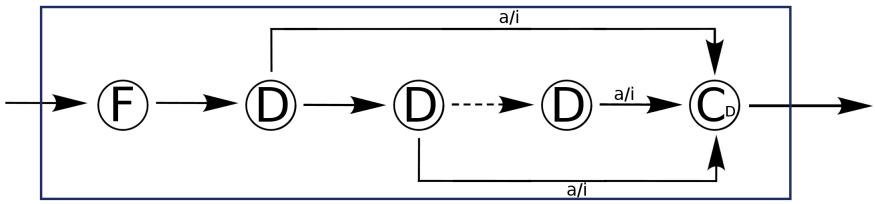


Figure 5: Generator of periodic patterns based on a series with multiple outputs.

Another example of pattern generator is illustrated in Figure 6.

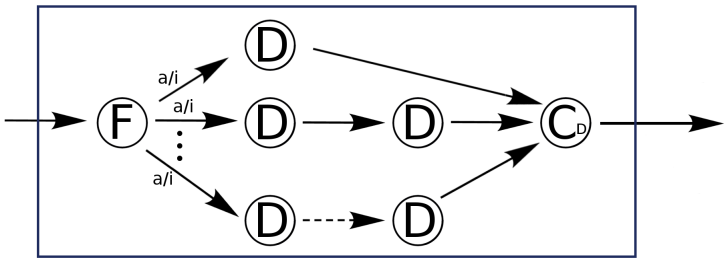


Figure 6: Generator of periodic patterns based on a parallel composition.

In this system, a $1/n$ filter is connected to $n$ simple series of delayers, of increasing length from 1 to $n$. The edges linking the filter to the series can be activated or deactivated (the key idea is that the activation of the series of length $x$ allows the emission of a 1 in the $x$-position of the pattern). While the first generator has a number of nodes and edges which is linear with respect to the length of the pattern to generate, in the second generator the number of nodes and edges is quadratic. For this reason, we employ the first generator for our studies.

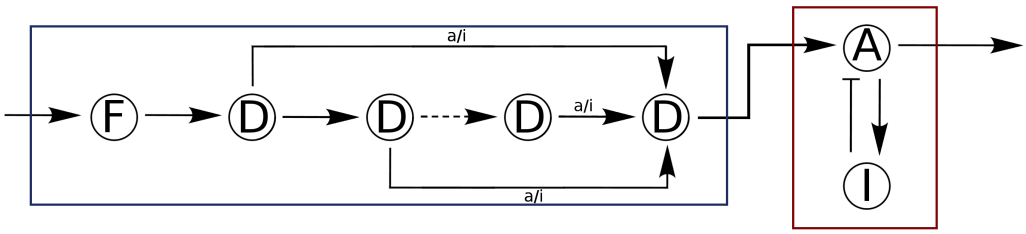


Figure 7: Generator of periodic patterns connected to the activator of a negative loop.

For different values of $n$, we connected the pattern generator to the input of the negative loop (see Figure 7) and we retrieved all the patterns able to produce oscillations. We take into account not only strict Square Oscillations, where the number of 1 equals the number of 0, but also Pulse Wave

Modulations with a ratio of almost fifty percent (where the difference between the consecutive number of 1 and the consecutive number of 0 is at most 2).

For instance, for $n = 5$, the pattern 11001 generates oscillations of the form 11000 as output of the negative loop (we recall that a simple series getting a sequence of 1 as input cannot emit a pattern of the form 11001).

### 6.3.4 Series within Contralateral Inhibition

In this subsection we study the integration of a simple series of $n$ delayers within a two neurons contralateral inhibition to defer the inhibition of the losing neuron. For this purpose, we use a first neuron $N_1$ acting as a delayer (if inhibitions are neglected) connected to a simple series $S_n$ of $n$ delayers, which inhibits a second neuron $N_2$. The second neuron (which is a delayer too) inhibits $N_1$ and the second inhibition is weaker than the first one (see Figure 8).

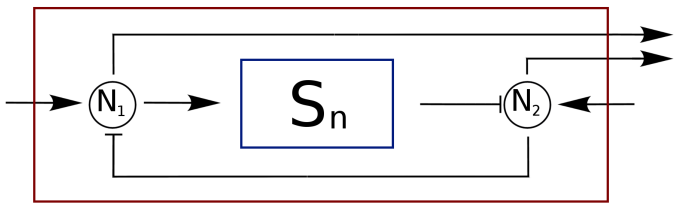


Figure 8: Series of $n$ delayers within a contralateral inhibition of two neurons.

The goal is to understand how the winner takes all behavior (see Property 7) is affected when the inhibition of the loser neuron is deferred. We verify that, for several inhibitor edge weights, the delayer series introduction makes the system stabilize later. Furthermore, as shown Figure 9, the stabilization is preceded by $n + 1$ damped oscillations of period $n + 2$.

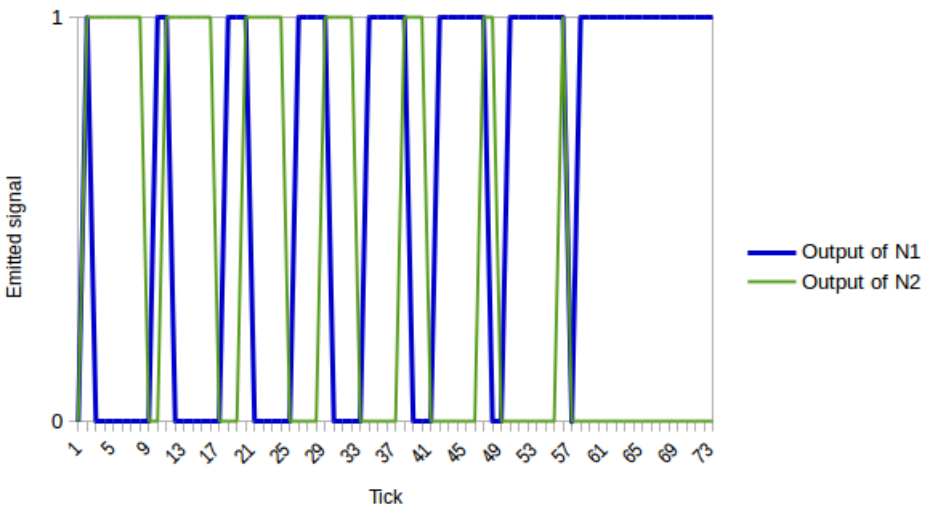


Figure 9: Output of N1 and N2 for a simple series of $n = 6$ delayers nested in a two neurons contralateral inhibition (see Figure 8).

The first oscillation of $N_1$ consists of one 1 and a number of 0 equal to $n+1$. For each subsequent oscillation, the number of 1 (resp. of 0) increases (resp. decreases) of one unit. After its last oscillation,

$N_1$ emits an infinite sequence of 1. The behavior of $N_2$ is symmetric (its first oscillation is composed of one 0 and $n+1$ occurrences of 1).

The following property has been modeled as a Lustre observer and proved for multiple inhibitor edge weights:

**Property 10** ***[Winner takes all delay in a series within a contralateral inhibition]*** *Let us consider a delayer neuron $N_1$ connected to a series of $n$ delayers $S_n$ inhibiting a delayer neuron $N_2$, which in turn inhibits $N_1$. The edge inhibiting $N_2$ has a higher absolute value than the one inhibiting $N_1$. Let us suppose a sequence of 1 is given as input of the archetype composition. When $N_1$ emits a sequence of 1 as long as the first sequence of 1 emitted by $N_2$, then, after the emission of a 0 by $N_1$, $N1$ (resp. $N_2$) only emits a sequence of 1 (resp. 0).*

The two diagrams of Figure 10 and Figure 11 allow to make the behavioral comparison between a two neuron contralateral inhibition and the same archetype equipped with a single delayer. In both plots, the y-axis (resp. x-axis) represents the weight of the edge inhibiting the first (resp. second) neuron. Blue (resp. red) points represent pairs of weight values for which the stabilization is (resp. is not) reached within the first four time units. We can identify that, passing from the first (Figure 10) to the second diagram (Figure 11), the red zone (non satisfaction of the winner takes all property) increases. Moreover, the growth of the red zone is asymmetric, which reflects the asymmetry of the archetype composition. Contrary to what is expected, the neuron proved to win more often within the first four time units is the one preceding the delayer series (even if its output inhibitor signal is delayed of one time unit).

Notice that the current observers we propose deal with infinite values but cannot cover the whole parameter space. In the future, we intend to define more subtle observers to improve the space covering. For instance, we plan to prove the stability (or the linear growth?) of the red regions of Figure 10.

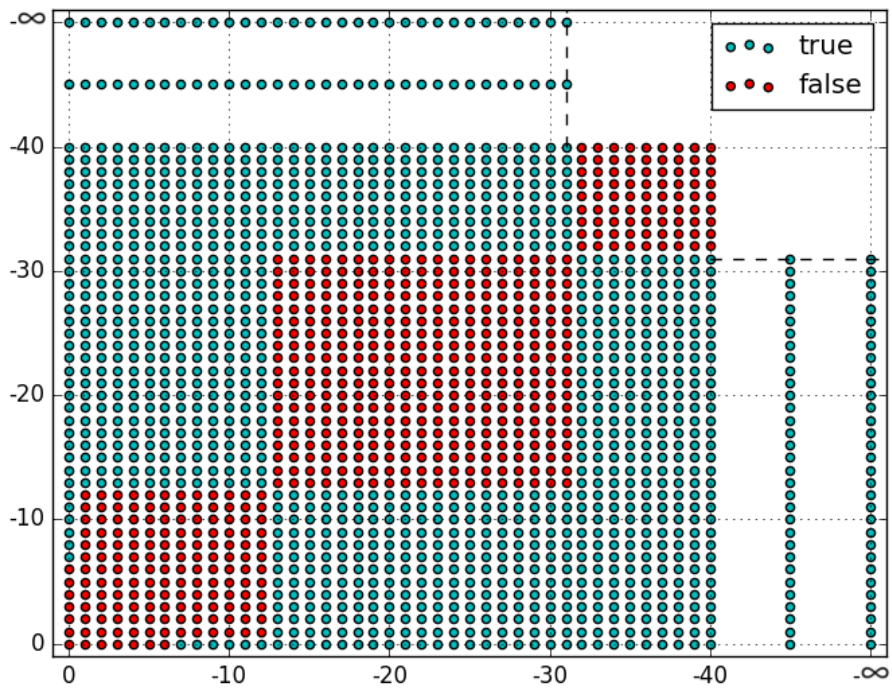


Figure 10: Verification of the winner takes all behavior for the different values of the inhibiting weights of the two neurons in a simple contralateral inhibition.

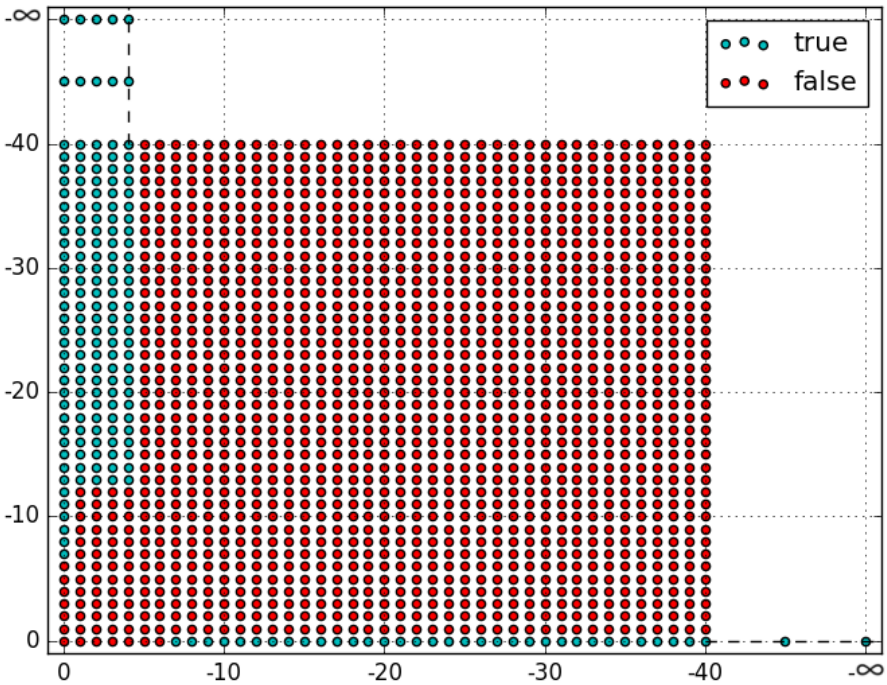


Figure 11: Verification of the winner takes all behavior for the different values of the inhibiting weights of the two neurons in a contralateral inhibition with two neurons and the insertion of one single delayer.

#### 6.3.5 Concatenation of Pattern Generator and Inhibition

As last representative example, we propose to concatenate the pattern generator with the inhibition archetype (see Figure 12).

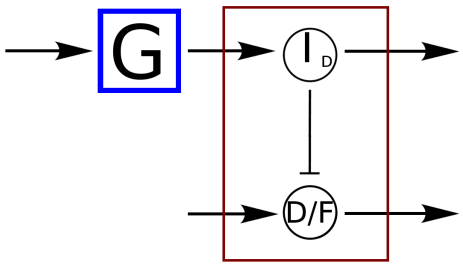


Figure 12: Pattern generator followed by inhibition.

As a first step, we kept the neuron parameters and edge weights obtained for the inhibition archetype alone, and searched for the input patterns able to entail the desired behavior (that is, starting from a given time, the inhibited neuron stops emitting spikes, see Property 6). With the previous parameters, the permanent sequence of 1 is the only pattern allowing to get the given behavior. We then succeeded in finding other patterns giving the inhibition property when the weight of the inhibiting edge is strengthened. We checked the property for every pattern of length $n < 8$. Patterns are classified with respect to the maximal inhibitor edge weight allowing to obtain the given behavior (histogram of Figure 13).

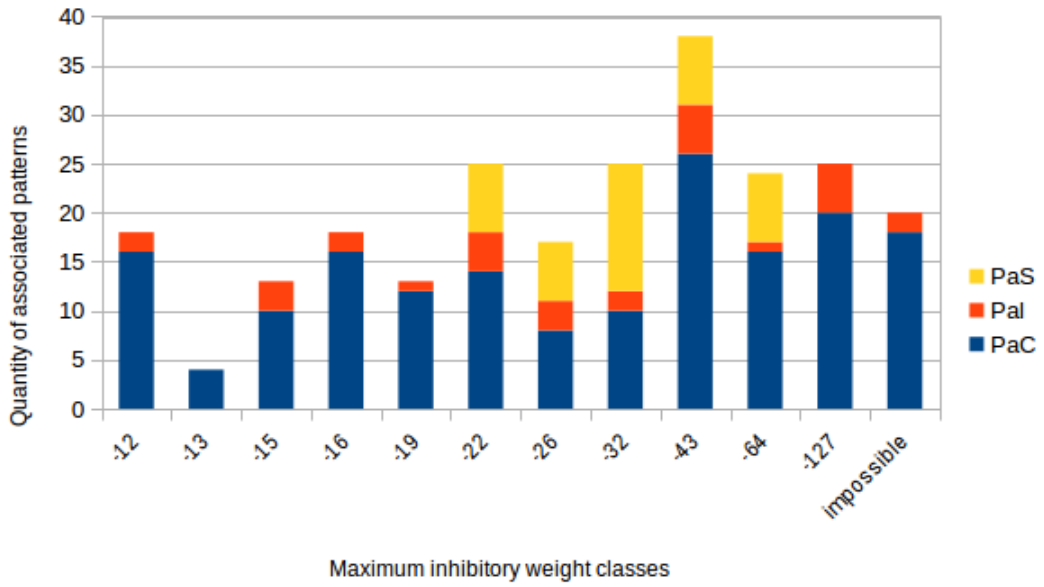


Figure 13: Classes of maximal inhibitory weights allowing the inhibition behavior. The orange color corresponds to palindrome patterns (Pal), the blue color corresponds to patterns whose twin is in the same class (PaC), and the yellow color stands for patterns whose twin falls in another class (PaS).

We can see that, if a given pattern is not a palindrome (that is, its reading from the left to the right and from the right to the left is different), it necessarily has a *twin pattern* (which is obtained by scanning the given pattern from the right to the left, and is different from the given pattern). There are several cases in which a pattern and its twin are not in the same weight class. For instance, for $n = 7$, the pattern 0110100 and its twin 0010110 are in two different (neighbor) classes. We can deduce that in a pattern, the number of zeros and ones is not the only important feature: the way zeros are interleaved by ones is relevant.

# 7 Theorem Proving Approach and Results

We first discuss the experimental setup in Section 7.1. Then in Section 7.2, we present the Coq definitions used to encode neurons and archetypes in Coq, and in Section 7.3, we discuss properties of archetypes and their proofs in Coq.

## 7.1 Experimental Setup

Our proof development was done in Coq version 8.11 on an Intel(R) Core(TM)-i7 4710HQ 2.50 GHz CPU with 8GB of RAM and on Windows 10.0 professional version. The Coq file takes 21.22 seconds to fully load and execute on this system. This file contains over 1,200 lines of Coq proof script; about 20% of the code defines the model and the rest contains the proofs. As described in the next subsection, the model is defined as a set of Coq definitions that allow for arbitrary values of parameters, such as any length of time, any input sequence, or any number of neurons in a simple series.

The real cost of proving theorems in a proof assistant is not the execution time needed to load and check the proofs nor the required memory, which is close to constant for our application; the real cost is the time that it takes for the user to interactively guide the proof assistant through the proof. Because of the complex nature of the properties we consider, automated theorem proving alone is not enough. The expert user must guide the proof assistant by choosing tactics that implement the high level ideas of the proof such as the kind of induction to use. The proof assistant can take care of many details automatically, but in our application, this occurs quickly enough so that an interactive user notices no delay. Once a proof is complete, it is checked and verified by the proof assistant. Any human mistakes result in an error report, which must be fixed before the system accepts the proof.

Although the exact time is hard to compute because the proofs were carried out over a long period of time, we estimate that on average it took one person approximately 2 full-time weeks to complete each proof. To give some examples, the delayer effect took about a month and the spike decreasing property required about a week. Since we use abstraction to break the development into lemmas, as we continue our development, we expect the difficulty of the interactive proof development to remain constant and the load time of the completed development to increase linearly with the number of lines of proof script.

### 7.2 Encoding Neurons and Archetypes in Coq

We start illustrating our formalization of neural networks in Coq with the code below.

```
Record Neuron := MakeNeuron {
  Output:list nat;
  Weights:list Q;
  Leak_Factor:Q;
  Tau:Q;
  Current:Q;
  Output_Bin: Bin_List Output;
  LeakRange: Qle_bool 0 Leak_Factor = true /\
             Qle_bool Leak_Factor 1 = true;
  PosTau: Qlt_bool 0 Tau = true;
  WRange: WeightInRange Weights = true }.

Fixpoint potential (Weights: list Q)
  (Inputs: list nat): Q :=
  match Weights, Inputs with
  | nil, _ => 0
  | _, nil => 0
  | h1::t1, h2::t2 => if (beq_nat h2 0%nat)
                      then (potential t1 t2)
                      else (potential t1 t2) + h1
  end.
```

To define a neuron, we use Coq's basic record structure. This record consists of five fields with their corresponding types, and four fields representing constraints that the first five fields must satisfy according to the LI&F model defined in Section 3. A neuron output `(Output)` is represented as a list of natural numbers, with one entry for each time step. The weights linked to the inputs of the neuron `(Weights)` are stored in a list of rational numbers (one for each input in some fixed order). The leak factor `(Leak_Factor)`, the firing threshold `(Tau)`, and the most recent neuron membrane potential `(Current)` are rational numbers. As far as the four conditions are concerned, `PosTau` states that `Tau` must be positive (i.e., `Qlt_bool 0 Tau = true` is the Coq representation for $0 < \tau$, which encodes the condition $\tau \in \mathbb{Q}^+$ from Definition 1). `Qlt_bool` and other arithmetic operators can be found in Coq's rational number library. The other three conditions state, respectively, that `Output` contains only 0s and 1s (it is a binary list), `Leak_Factor` is between 0 and 1 inclusive, and each input weight is in the interval [-1, 1]. We do not provide the definitions of `Bin_List` and `WeightInRange` used in these statements. The reader can consult the accompanying Coq code for details.

For each neuron `N`, we write `(Output N)` to denote its first field, and similarly for the other fields. To define a new neuron with values `O, W, L, T,` and `C` with the suitable types, and proofs `P1,...,` `P4` of the four constraints, we write `(MakeNeuron O W L T C P1 P2 P3 P4)`.

The next definition in the above code computes the weighted sum of the inputs of a neuron, which is fundamental for the calculation of the potential function of a neuron (see Definition 1). In this recursive function, there are two arguments: `Weights`, which represents some number $m$ of weights

$w_1, \ldots, w_m$, and `Inputs`, which represents $m$ inputs $x_1, \ldots, x_m$. The function returns an element of type `Q`. Its definition employs pattern matching on both inputs at the same time. In the body of the definition there are booleans, the `if` statement, and the equality operator on natural numbers `(beq_nat)`, all from Coq's standard libraries. Natural number constants, such as `0%nat` above, are given with their types to differentiate them from rational number constants, whose types are omitted. Although, the `potential` function is always called with two lists of the same length, Coq requires functions to be total; if two lists have different length, we return a "default" value of 0 in the base cases. Furthermore, when this function is called, `Inputs`, which is the second argument of the function, must be a binary list (that is, it can only contain the natural numbers 0 and 1). Thus, when the head of the list `h2` is 0, we do not need to add anything to the final sum because anything multiplied by 0 is 0. In such a case, we just call the function recursively on the remaining weights and inputs `t1` and `t2`, respectively. On the other hand, if `h2` is 1, we add `h1`, the head of `Weights`, to the final sum, which is the recursive call on `t1` and `t2`. We need to implement the potential function in this way because `h1` and `h2` cannot be multiplied in Coq because they have different types. Recall that `h1` is a rational number and `h2` is a natural number.

The following code illustrates the `NextPotential` function, which computes $p(t)$ from Definition 1.

```
Definition NextPotential
 (N: Neuron) (Inputs: list nat): Q :=
 if (Qle_bool (Tau N) (Current N))
 then (potential (Weights N) Inputs)
 else (potential (Weights N) Inputs) +
      (Leak_Factor N) * (Current N).
```

Recall that `(Current N)` is the most recent potential value of the neuron, which is $p(t-1)$ in Definition 1. `(Qle_bool (Tau N) (Current N))` represents $\tau \leq p(t-1)$, and we use the potential function defined before to compute the weighted sum of the neuron inputs. Finally, the last line computes $r \cdot p(t-1)$.

The following code gives two important definitions.

```
Definition NextOutput
  (N: Neuron) (Inputs: list nat): nat :=
  if (Qle_bool (Tau N) (NextPotential N Inputs))
  then 1%nat
  else 0%nat

Definition NextNeuron
  (N: Neuron) (Inputs: list nat): Neuron :=
  MakeNeuron
    ((NextOutput N Inputs)::(Output N))
    (Weights N)
    (Leak_Factor N)
    (Tau N)
    (NextPotential N Inputs)
    (NextOutput_Bin_List N Inputs (Output_Bin N))
    (LeakRange N)
    (PosTau N)
    (WRange N).
```

The first definition calculates the next output of the neuron, which is $y(t)$ in Definition 1. Recall that `(NextPotential N Inputs)` computes $p(t)$. The expression `(Qle_bool (Tau N) (NextPotential N Inputs))` thus encodes the constraint $\tau \leq p(t)$.

In our model, the state of each neuron is represented by the `Output` and `Current` fields. The `Output` field of a neuron in the initial state is `[0%nat]`, which represents a list containing one 0. The `Current` field denotes the initial potential value, which is set to 0. A neuron changes its state by processing its inputs. After treating a list of $n$ inputs, the `Output` field becomes a list of length $n+1$ containing 0's and 1's, and the `Current` field is set to the value of the potential after processing these $n$ inputs. A state change occurs by applying the `NextNeuron` function reported in the above code to a neuron and a list of inputs. We represent a neuron at its later state by generating a new record with the new values for `Output` and `Current`, and other values directly copied over. We store the values in the `Output` field in reverse order, which helps making proofs by induction over lists easier in Coq. Thus, the most recent output of the neuron is at the head of the list. This can be seen in the above code, where the new value of the output is `((NextOutput N Inputs)::(Output N))`. The next output of the neuron is at the head, and is followed by the previous outputs. `(NextPotential N Inputs)` is the new value for `(Current N)`. Recall that `(Current N)` is the most recent value of the potential value of the neuron, or $p(t-1)$. So, for computing the next potential value of the neuron or $p(t)$, the `NextPotential` function must be called.

Proofs of the four constraints result from the new values for each field of the neuron. The first one requires a lemma `NextOutput_Bin_List` (statement omitted) proving that the new longer list is still a binary list. Proofs of the other three constraints are carried over exactly from the original neuron, since they concern some components of the neuron that remain unchanged.

The `ResetNeuron` function is employed to reinitialize a neuron to the initial state.

```
Definition ResetNeuron
  (N: Neuron): Neuron := MakeNeuron
  ([0%nat])
  (Weights N)
  (Leak_Factor N)
  (Tau N)
  (0)
  (Reset_Output)
  (LeakRange N)
  (PosTau N)
  (WRange N).
```

This function takes any `Neuron` as input, and returns a new one, where the `Output`, `Current`, and `Output_Bin` fields are reset, while the other fields are unchanged. The `Reset_Output` property is a simple lemma stating that `[0%nat]` satisfies the `Bin_List` property.

So far, we have presented the encoding of single neurons in isolation from other neurons. We next consider archetypes. In general, our approach is to encode the particular structure of each archetype as a Coq record. Using a record for each archetype facilitates stating and proving properties about them. Recall that archetypes are functional structures of neural networks. Defining them in this abstract way helps us to present their basic functions. To illustrate this approach, we introduce now the encoding of two archetypes, the simple series in Figure 1(a) and the negative loop in Figure 1(d). As shown in Figure 1(a), a simple series consists of a list of single input neurons. The first neuron receives the input of the archetype and sends its output to the second neuron. Starting from the second neuron each neuron receives its input from the previous neuron and sends its output as the input of the next neuron in the series. The last neuron produces the output of the series. The `NeuronSeries` record defined below represents this structure in Coq.

```
Record NeuronSeries {Input: list nat} :=
MakeNeuronSeries
{
  NeuronList: list Neuron;
  NSOutput: list nat;
  AllSingle: forall (N:Neuron),
    In N NeuronList ->
    (beq_nat (length (Weights N)) 1%nat) = true;
  SeriesOutput: NSOutput =
  (SeriesNetworkOutput Input NeuronList);
}.
```

Records can have input parameters, similar to functions in Coq, and here the list of inputs to the simple series is `Input: list nat`. Thus `NeuronSeries` is actually a function from a list of natural numbers to a record. Curly brackets around input arguments is Coq notation for *implicit* arguments, which are arguments that can be omitted from expressions as long as Coq can figure out the missing information. Its use here allows us to write more readable Coq code. `NeuronList` is a field in the record representing the list of neurons in the simple series. The first element in this list is the first neuron is the series, etc. `NSOutput` represents the list of outputs of the series. In other words, it is the output list of the last neuron in the series, which is also the last neuron of `NeuronList`. There are also two constraints for this archetype. `AllSingle` expresses that all neurons in the series are single input neurons. The functions `In` and `length` are defined in Coq's list library and define list membership and size of a list, respectively. `SeriesOutput` expresses that the output of the series is equal to the output of the function `SeriesNetworkOutput`. This function takes the input of the series and list of neurons in the series and produces the output of the series. We leave out its definition and just note here that it expresses the details of the input/output connections between the elements of `NeuronList`, and in the degenerate case when `NeuronList` is empty, `NSOutput` is set to the input. (See the definition in our accompanying code.)

The Coq definition of the negative loop is shown below.

```
Record NegativeLoop {Inputs: list nat} :=
MakeNegativeLoop {
  N1: Neuron;
  N2: Neuron;
  NinputN1: (beq_nat (length (Weights N1)) 2%nat)
            = true;
  NinputN2: (beq_nat (length (Weights N2)) 1%nat)
            = true;
  PW1: 0 < (hd 0 (Weights N1));
  PW2: (hd 0 (tl (Weights N1))) < 0;
  PW3: 0 < (hd 0 (Weights N2));
  Connection1: Eq_Neuron2 N1
               (AfterNArch2N1 (ResetNeuron N1)
                              (ResetNeuron N2)
                              Inputs);
  Connection2: Eq_Neuron2 N2
               (AfterNArch2N2 (ResetNeuron N1)
                              (ResetNeuron N2)
                              Inputs)
}.
```

There are only two neurons in this archetype. These neurons are represented by `N1` and `N2` in the definition. The rest of the fields are constraints defining the properties and connections of the

archetype. `NinputN1` expresses that `N1` has two inputs. Similarly, `NinputN2` states the number of inputs for `N2`. `N2` is a single input neuron. As mentioned earlier, in Figure 1(d), the solid black circle for an input arrow means that the input has a negative weight and the absence of this circle means that the input has a positive weight. These properties are expressed by constraints `PW1` and `PW2`. In particular, the first input acts as an activator for `N1` and the second one is an inhibitor for `N1`. `PW3` expresses that the only input of `N2` has a positive weight and so is an activator for `N2`. `Connection1` defines the connection of the output of `N2` to the second input of `N1` using the `AfterNArch2N1` function. This function returns a neuron that represents the status of `N1` after applying all inputs in the input list. Again, we leave out the details of the formal definition and refer the reader to the accompanying Coq code. We simply note here that in order to compute this result, three arguments are required—the initial status of `N1`, the initial status of `N2`, and the list of inputs to the archetype. Similarly to the definition of `NeuronSeries`, `Inputs` is an argument of the record `NegativeLoop`. The `Eq_Neuron2` function (whose definition is also omitted, see the code) checks equality of neurons by checking equality of each individual field. `Connection2` defines the connection of the output of `N1` to the only input of `N2` and is defined similarly to `Connection1`. `Connection2` uses `AfterNArch2N2`, which is the same as `AfterNArch2N1` except that it returns `N2` after applying all inputs in the input list.

### 7.3 Properties of Archetypes and their Proofs in Coq

Some of the properties presented in Section 6.2 concerning neuron and archetype behaviors have already been fully verified in Coq. The most important ones are: the delayer and filter effects for a single neuron (Property 1), the $n$-delayer effect for a simple series (Property 2), and fixed-point inhibition for an inhibition archetype (Property 6). For illustration, we report on the Coq proofs of the first two, plus provide one supplementary property. We give their complete proofs to show the structure of the main inductions as well as to show other high-level mathematical proof strategies used in the proofs. Many such strategies can be mapped to Coq proof commands called *tactics*; some examples that can be seen in the code include, `induction`, `inversion`, and `destruct`.

In the statement of the properties we present, we omit the assumption that the input sequence of the neuron is a binary list containing only 0s and 1s. It is, of course, taken into account in the Coq code. We adopt many other conventions to enhance readability when stating properties and presenting the corresponding proofs. We state the properties using pretty-printed Coq syntax, with some abbreviations for our own definitions. For instance, we use mathematical fonts and conventions for Coq text, e.g., `(Output N)` is written $Output(N)$, `(Tau N)` is written $\tau(N)$, `(Weights N)` is written $w(N)$, `(Leak_Factor N)` is written $r(N)$, and `(Current N)` is written $p(N)$. In addition, if $w(N)$ is a list of the form $[w_1; \ldots; w_n]$ for some $n \geq 0$, for $i = 1, \ldots, n$, we often write $w_i(N)$ to denote $w_i$. Furthermore, we use notation and operators from the Coq standard library for lists. For example, *length* and $+$ are list operators; the latter is the notation used here for list concatenation. In addition, although for a neuron $N$, the list $Output(N)$ is encoded in reverse order in our Coq model, we use forward order when presenting properties and proofs here.

#### 7.3.1 The Delayer Effect for a Single-Input Neuron

Recall that the delayer effect of Property 1 of Section 6.2.1 concerns a single neuron having only one input. Since a neuron is in an inactive state initially, its output at time 0 is 0. When a neuron has only one input, and the corresponding weight is greater than or equal to the neuron activation threshold, then the neuron transfers the input sequence to the output without any modification (except for a "delay" of length 1). For example, if a single input neuron receives 0100110101 as its input sequence, it will produce 00100110101 as output. Neurons with this property are mainly just transferring signals. Humans have some of this type of neuron in their auditory system, associated to a chemical synapse. This property (corresponding to the delayer effect of Property 1) is formalized as Property 11.

**Property 11** ***[Delayer effect for a single-input neuron]***

$$\forall (N : neuron)(input : list\ nat),$$
$$length(w(N)) = 1 \wedge w_1(N) \geq \tau(N) \rightarrow$$
$$Output(N') = [0] + input$$

In the above statement, $N'$ denotes the neuron obtained by initializing $N$ and then processing the input (using $ResetNeuron$ and repeated applications of $NextNeuron$). Observe that in Definition 1, $p$ is a function of time. Time in our Coq model corresponds to the position in the output list. If $Output(N)$ has length $t$, then $p(N)$ stores $p(t-1)$ from Definition 1. By applying $NextNeuron$ to $N$ and to the next input obtaining $N'$, we obtain that $Output(N')$ has length $t+1$ and $p(N')$ stores the value $p(t)$ from Definition 1.

In order to prove Property 11, we require the following lemma, which states that when a neuron has one input and the input weight is greater than or equal to the threshold, then the potential value of that neuron is always non-negative.

**Lemma 1**

$$\forall (N : neuron)(input : list\ nat),$$
$$length(w(N)) = 1 \wedge w_1(N) \geq \tau(N) \rightarrow p(N') \geq 0$$

As previously explained, $p(N')$ is the most recent value of the potential function of neuron $N$, i.e., the one obtained after processing all the input values. The proof of this lemma is in the accompanying Coq code. We use it here to prove Property 11.

**Proof 1 (of Property 11)** *The proof is by induction on the length of the input sequence as follows.*

*Base case: $input = []$ (the empty list). If there is no input in the input sequence, the neuron will keep its initial status, i.e., $N = N'$. So, $Output(N') = [0]$. Therefore, $Output(N') = [0] = [0] + [] = [0] + input$.*

*Induction case: We assume that the property is true for input and we must show that it holds for some $input'$ of the form $(input + [h])$ for some additional input value $h$. Let $N'$ be the neuron resulting from processing input, and let $N''$ be the neuron after processing $input'$. By the induction hypothesis, we know $Output(N') = [0] + input$ and we must prove that $Output(N'') = [0] + input'$.*

*Note that $\tau(N) = \tau(N') = \tau(N'')$ and similar equalities hold for $r$ and $w_1$, so we use them interchangeably. Because input is a binary list, we know that $h = 0$ or $h = 1$. We break the proof into two different cases, depending on the value of $h$.*

*First, we assume that $input' = input + [0]$ and we prove that $Output(N'') = Output(N') + [0]$. In this case, the most recent input to the neuron is 0. Again, to relate this to Definition 1, let $t$ be the time at which we process the last input. We calculate $p(N'')$, which corresponds to $p(t)$, i.e., the potential value of the neuron at time $t$; also the value $p(N')$ represents $p(t-1)$ in this definition. Using the first and second clauses of the definition of $p(t)$, respectively, the value is one of:*
$p(N'') = w_1(N') \cdot 0 = 0$ *or*
$p(N'') = w_1(N') \cdot 0 + r(N') \cdot p(N')$.

*In the first case, $p(N'') = 0$ and we know $0 < \tau(N)$, because $\tau(N)$ is always positive. So, by the second clause of the definition of $y(t)$, the next output of the neuron will be 0. The other case, which comes from the second clause of the definition of $p(t)$, has the same result. In this case, the condition on this clause says that $p(N') < \tau(N')$ and we must show that $p(N'') = r(N') \cdot p(N') < \tau(N)$. Recall that $r(N')$, the leak factor of the neuron, is between 0 and 1. So, multiplying any number, that is less than a positive number, by a value between 0 and 1, gives a value that is smaller than or equal to the original number. Therefore, by the definition of $y(t)$, the next output of the neuron will be 0 again. We can conclude now that by adding 0 to the input sequence, a 0 will be produced in the output. Thus, $Output(N'') = Output(N') + [0]$. Using our induction hypothesis, we have: $Output(N'') = Output(N') + [0] = [0] + input + [0] = 0 + input'$.*

*Second, we assume that* $input' = input + [1]$ *and we will prove that* $Output(N'') = Output(N') + [1]$. *In this case, the most recent input of the neuron is 1. Again, we calculate the potential value of* $N''$ *the definition of* $p(t)$*:*
$p(N'') = w_1(N') \cdot 1 = w_1(N')$ *or*
$p(N'') = w_1(N') \cdot 1 + r(N') \cdot p(N') = w_1(N') + r(N') \cdot p(N')$.

*In the first case, when* $p(N'') = w_1(N')$, *we know that* $w_1(N) \geq \tau(N)$ *by assumption in the statement of the property; we know that* $w_1(N) = w_1(N')$ *as discussed, and thus* $p(N'') \geq \tau(N)$. *So by the definition of* $y(t)$, *the next output of the neuron will be 1. In the second case,* $p(N') \geq 0$ *according to Lemma 1, and it is always the case that* $r(N') \geq 0$, *so we can conclude that* $r(N') \cdot p(N') \geq 0$. *Because* $w_1(N) \geq \tau(N)$ *and adding a non-negative value to the greater side of an inequality keeps it that way, we can conclude that* $p(N'') = w_1(N') + r(N') \cdot p(N') \geq \tau(N)$. *Therefore, again by the definition of* $y(t)$, *the next output of the neuron will be 1 again. Thus, we can conclude in both cases that by adding 1 to the input sequence, a 1 will be produced in the output. Thus,* $Output(N'') = Output(N') + [1]$. *Using our induction hypothesis, we have:* $Output(N'') = Output(N') + [1] = [0] + input + [1] = 0 + input'$.

*This completes the proof.*

### 7.3.2 The Delayer Effect for a Simple Series

Here we consider the $n$-delayer effect for the simple series in Figure 1(a) (Property 2 of Section 6.2.2). If we have a series of $n$ single input neurons and all of them have the delayer effect, then the output of the whole structure is the input plus $n$ leading zeros. In other words, this structure transfers the input sequence exactly with a delay marked by the $n$ leading zeros, denoted as $zeros(n)$ in the statement of the property below. The `NeuronSeries` record defined in Section 7.2 to represent a simple series is denoted *NeuronSeries* here. This property (corresponding to the $n$-delayer effect of Property 2) is expressed as follows.

**Property 12** ***[Delayer effect for a simple series]***

$$\begin{gathered}\forall (input : list\ nat)(Series : (NeuronSeries\ input))(n : nat),\\ length(NeuronList) = n \wedge\\ \forall\ i = 1, \ldots n,\ length(w(NeuronList[i])) = 1 \wedge\\ \forall\ i = 1, \ldots n,\ w_1(NeuronList[i]) > \tau(NeuronList[i]) \rightarrow\\ Output = zeros(n) + input\end{gathered}$$

In this statement, *Series* is a variable of record type $(NeuronSeries\ input)$, where *input* is the argument to the *NeuronSeries* function, which builds a record. For readability, we abbreviate $(NeuronList\ Series)$, which represents the first field of *Series*, as $NeuronList$. Also, $NeuronList[i]$ denotes the $i^{th}$ neuron in *NeuronList* and *Output* abbreviates $(NSOutput\ Series)$, which is equal to $(Output\ (NeuronList[n]))$

**Proof 2 (of Property 12)** *The proof this time is by induction on the length of* $NeurlonList$.

*Base case:* $N = 0$. *This is the degenerate case where there are no neurons in the series and the output of the series is set to the input. (In particular, recall that the* `SeriesOutput` *constraint of the* `NeuronSeries` *record is defined using the* `SeriesNetworkOutput` *function, which is defined so that it returns the input list.) Thus in this case,* $Output = input = zeros(0) + input$.

*Base case:* $N = 1$. *In this case, there is only one neuron in the series and we know that this neuron has the delayer effect. According to Property 11 proved earlier, we can conclude that* $Output = [0] + input = zeros(1) + input$.

*Induction case: We assume that the property holds for* $NeuronList$ *of length* $k$. *Let* $Output'$ *be the output of this series of length* $k$. *Thus by the induction hypothesis,* $Output' = zeros(k) + input$. *We must show that the property holds for a* $NeuronList + [M]$, *where* $M$ *is a neuron such that* $length(w(M)) = 1$ *and* $w_1(M) > \tau(M)$. *Let* $Output''$ *be the output of this series of length* $k + 1$. *The input sequence for* $M$ *is the final output of* $NeuronList$, *which is* $zeros(k) + input$. *By the assumptions of this property, all neurons in* $NeuronList + [M]$ *satisfy Property 11, i.e., have the*

*delayer effect, including the last one $M$, which means that its output is equal to its input plus a leading 0. In other words, $Output'' = [0] + zeros(k) + input$. Therefore, we can conclude that $Output'' = [0] + zeros(k) + input = zeros(k+1) + input$.*

*This completes the proof.*

### 7.3.3 The Spike Decreasing Property for a Single-Input Neuron

The *spike decreasing* property states that a single input neuron cannot produce in its output sequence more 1s than the number of 1s in its input sequence. For example, if the input sequence is 11100110101, which contains seven 1s, the output of the neuron will have less than or equal to seven number of 1s. This property is a consequence of the fact that a single input neuron has either the delayer effect or the filter effect, and both of them do not increase the number of 1s. It is an important property of LI&F neurons and is expressed as property 13.

**Property 13** ***[Spike decreasing property for a single-input neuron]***

$$\forall (N : neuron)(input : list\ nat),$$
$$length(w(N)) = 1 \rightarrow$$
$$count(input, 1) \geq count(Output(N'), 1)$$

In this property, *count* is a function that calculates the number of occurrences of the number given as the second argument in the list given as the first argument. So, $count(input, 1)$ returns the number of 1s in the input list and $count(Output(N'), 1)$ computes the number of 1s in the output list of the neuron $N'$. Recall that here $N'$ is the neuron after initializing $N$ and then applying all the inputs in the input list.

A lemma is needed to prove this property. The following lemma expresses that when a single input neuron receives a 0 as input at any point in time, it cannot produce 1 in the output as follows.

**Lemma 2**

$$\forall (N : neuron)(input : list\ nat),$$
$$length(w(N)) = 1 \wedge input = [0] \rightarrow last(Output(N')) = 0$$

In the statement of this lemma, *last* represents the last element of a list and $N'$ represents the neuron obtained from $N$ by processing a single 0 as the next input. (In other words, $N'$ is obtained from $N$ by a single application of $NextNeuron$ without first applying $ResetNeuron$.) Thus, $p(N')$ represents the value used to calculate $last(Output(N'))$, which is the last output value of the neuron. The proof of this lemma is a straightforward consequence of Definition 1.

**Proof 3 (of Lemma 2)** *Since the input is a single 0, we know from Definition 1 that $p(N') = w_1(N) \cdot 0$ or $p(N') = w_1(N) \cdot 0 + r(N) \cdot p(N)$. Simplifying the multipication by 0, we have these two cases: $p(N') = w_1(N) \cdot 0 = 0$ or $p(N') = w_1(N) \cdot 0 + r(N) \cdot p(N) = r(N) \cdot p(N)$. We know that $\tau(N') = \tau(N) > 0$. Thus, in the first case, $p(N') = 0 < \tau(N')$. According to Definition 1, the next output of the neuron will be 0 because its potential value $p(N')$ is less than its activation threshold $\tau(N')$.*

*In the second case, when $p(N') = r(N) \cdot p(N)$, we know that $p(N) < \tau(N) = \tau(N')$. Also, for every neuron, the leak factor is between 0 and 1. In other words, $0 \leq r(N) = r(N') \leq 1$. If $p(N) < 0$, multiplying it by a positive number will produce a negative value which is less than $\tau(N)$. So, $p(N') = r(N) \cdot p(N) < 0 < \tau(N) = \tau(N')$. On the other hand, if $p(N) \geq 0$, multiplying it by a number between 0 and 1, will produce a smaller number. So, $p(N') = r(N) \cdot p(N) < p(N) < \tau(N) = \tau(N')$. As can be seen, $p(N') < \tau(N')$ in this case also and thus the next output of the neuron will be 0. This completes the proof.*

We can now use this lemma to prove property 13.

**Proof 4 (of Property 13)** *The proof is by induction on the length of the input sequence as follows.*

*Base case: input = [] (the empty list). If there is no input in the input sequence, the neuron will keep its initial status, i.e.,* $N = N'$*. So,* $Output(N') = [0]$*. Therefore,* $count(input, 1) = count([], 1) = 0 \geq count(Output(N'), 1) = count([0], 1) = 0$*.*

*Induction case: We assume that the property is true for input and we must show that it holds for some* $input'$ *of the form* $(input + [h])$ *for some additional input value* $h$*. Let* $N'$ *be the neuron resulting from processing input, and let* $N''$ *be the neuron after processing* $input'$*. By the induction hypothesis, we know* $count(input, 1) \geq count(Output(N'), 1)$ *and we must prove that* $count(input', 1) \geq count(Output(N''), 1)$*.*

*Because input is a binary list, we know that* $h = 0$ *or* $h = 1$*. We break this into two different cases, depending on the value of* $h$*.*

*First, we assume that* $input' = input + [0]$*. Because 0 is added to the input list, we have* $count(input', 1) = count(input, 1)$*. According to Lemma 2, that we just proved, having 0 as the last input in* $input'$ *cannot produce 1 as the last output in the output list. So, we can conclude that* $Output(N'') = Output(N') + [0]$*. Similarly, because 0 is added to the output,* $count(Output(N'), 1) = count(Output(N''), 1)$*.*

*Therefore,* $count(input', 1) = count(input, 1) \geq count(Output(N'), 1) = count(Output(N''), 1)$*.*

*Second, we assume that* $input' = input + [1]$*. In this case, 1 is added to the input list. Because the number of 1s is increased, it is not important that this new input produce 0 or 1 in the output list. The number of 1s in the output list will remain unchanged or increase by 1. So,* $count(Output(N''), 1)$ *is at most* $count(Output(N'), 1) + 1$*. Also, we know that* $count(input', 1) = count(input + [1], 1) = count(input, 1)+1$*. By the induction hypothesis we know that,* $count(input, 1) \geq count(Output(N'), 1)$*. By adding 1 to both sides of the inequality, we will have* $count(input, 1)+1 \geq count(Output(N'), 1)+1$*. Therefore,* $count(input, 1) + 1 = count(input', 1) \geq count(Output(N''), 1)$*.*

*This completes the proof.*

# 8 Comparison of the Proposed Approaches and Future Work

In this paper we proposed two formal approaches to study some key properties concerning the dynamic evolution of neurons, some basic archetypes, and some archetype couplings. These formal approaches are original and complementary with respect to the main international projects aiming at understanding the human brain, such as the Human Brain Project [73], which is mainly based on large systems of differential equations.

We provide a full implementation for both approaches. As far as the model checking approach is concerned, the Lustre code for all the archetypes and properties can be found at

`https://redmine.i3s.unice.fr/projects/archetypes/repository`.

As far as the theorem proving approach is concerned, as mentioned, the Coq code for the archetype implementation, properties, and proofs is given at

`http://www.site.uottawa.ca/~afelty/coq-archetypes/`.

First of all, we should underline that both approaches are likely to be improved. Concerning the model checking technique, before checking the validity of a property, we actually fix an interval of values for each parameter and we ask to the model checker whether the property holds for the chosen parameter set. We are aware of a research group working on the integration of Linear Decision Diagrams (LDD) within the model checker. This would allow to automatically infer parameter sets for which properties are verified. Concerning the theorem proving technique, although the proofs we have completed require some sophisticated reasoning, there is still a significant amount that is common between them. As we continue, we expect to encounter more complex inductions as we consider more complex properties. Thus, it will become important to automate as much of the proofs as possible, most likely by writing tactics tailored to the kind of induction, case analysis, and mathematical reasoning that is needed here. Furthermore, defining general relations that can be specialized to

| **Model Checkers** | |
|---|---|
| + | Press-button approach |
| + | Counter-example automatically provided |
| - | Prove properties for some given parameter values |
| **Theorem Provers** | |
| - | An expert is needed |
| + | Successful proofs help in proving other properties |
| + | Prove properties at a high level of generality |

Table 1: Pros and cons of model checkers and theorem provers.

specific patterns will likely also be very useful for the kinds of properties that are important for more complex networks.

In absolute terms, we could not say one of the two approaches presented here is strongly preferable with respect to the other one for the formal study of the dynamic properties of neuron archetypes. The main advantage of the model checking methodology is that it is completely automatic: once a given property has been correctly encoded, the user just needs to press a button to know whether the property is verified or not. So the presence of an expert is not needed to obtain a proof. Another strength of model checking is that, in case a property does not hold in a given model, a counter-example is automatically provided. Such an execution trace can give hints in understanding what should be modified in the system so that the property is satisfied. As another advantage, we should underline that for our application domain model checking is not time consuming. In fact, even if the transition system corresponding to each model is exponential with respect to the number of variables, we could get an immediate answer for all our properties because the chosen model checker, kind2 [26], exploits some advanced features to improve scalability, such as *modular reasoning* (each node can be assigned its own properties and verified individually. The results of the verification process can be reused in the analysis of other components calling that node).

On the other hand, model checkers often cannot prove properties at the desired level of generality. As a matter of fact, the use of a proof assistant guarantees that the properties we prove are true in the *general case*, such as true for any input values, any length of input, and any amount of time. As an example, let us consider the simple series. With Lustre, we were able to write a node which encodes the expected behavior of the circuit. Then, we could call the kind2 model checker to test whether the property at issue is valid for some input series with a fixed length. With Coq we can prove that the desired behavior is true whatever the length and the parameters of the series are. Another advantage of the theorem proving approach is that, since we have access to proofs, a successful proof of a given property can be exploited to prove similar properties. The drawbacks are that proofs can be long and an expert is needed to perform them. The main advantages (+) and drawbacks (-) of model checkers and theorem provers are summarized in Table 1.

Coming back to the biological issue that motivated our study, our results until now show that archetype coupling can either *modulate* the behaviors displayed by the single archetypes (e.g. extend an oscillation period) or clearly give rise to *new behaviors*. Furthermore, several different couplings turn out to display the same behavior (whatever their input sequences are). As a next step, we intend to make a systematic study of all the possible archetypes and couplings, even including several archetypes, and classify their respective properties. The number of constrained graphs of a few elements and, concomitantly, the number of elementary behaviors is reduced, both from a logical and biological standpoint. To use again the analogy of words, few syllables allow to build thousands of currently used words and a quasi-infinity of sentences. Moreover, beyond a certain number of neurons and above all a certain level of connectivity, other functional processes different from archetypes are supposed to emerge, like cell assemblies [74, 75]. So the composition process should stop after a finite and limited (compared to the size of the brain) number of iterations. We should thus rapidly fall back on already studied behaviors, but expressed through various instances of neuronal networks, reducible to few

neuronal archetypes. This means that, at a certain point, this approach should allow to express any relevant neuronal network as an archetype (a basic one or the result of a concatenation procedure).

Our feeling is that, to perform a formal advanced analysis of archetypes and their composition, the model checking and theorem proving approaches should be used together in a pragmatic way. When trying to prove a given property, the idea is to first test its validity for some crucial given parameter intervals using model checking, and eventually refine the model thanks to the provided counter-examples so that the property holds in the defined context. Once some key tests have passed, the theorem proving technique can be exploited to prove the property in a more general context. The complementarity of these two powerful techniques could make us advance rapidly in the study of the fundamental structural and functional properties of the elementary building blocks of the brain and cognition.

**Acknowledgements** This work was supported by the French government through the UCA-Jedi project managed by the National Research Agency (ANR-15- IDEX-01) and, in particular, by the interdisciplinary Institute for Modeling in Neuroscience and Cognition (NeuroMod) of the Université Côte d'Azur. It was also supported by the Natural Sciences and Engineering Research Council of Canada. We thank Alexandre Muzy for fruitful discussions on neuron modeling.

## References

[1] Olaf Sporns. The human connectome: Origins and challenges. *NeuroImage*, 80:53 – 61, 2013. Mapping the Connectome.

[2] Olaf Sporns. Structure and function of complex brain networks. *Dialogues in clinical neuroscience*, 15:247–262, 2013.

[3] Olaf Sporns. Graph theory methods: applications in brain networks. *Dialogues in clinical neuroscience*, 20:111–121, 2018.

[4] Alex Fornito, Andrew Zalesky, and Michael Breakspear. Graph analysis of the human connectome: Promise, progress, and pitfalls. *NeuroImage*, 80:426 – 444, 2013. Mapping the Connectome.

[5] Kiyotoshi Matsuoka. Mechanisms of frequency and pattern control in the neural rhythm generators. *Biological cybernetics*, 56(5-6):345–353, 1987.

[6] Régis C. Lambert, Christine Tuleau-Malot, Thomas Bessaih, Vincent Rivoirard, Yann Bouret, Nathalie Leresche, and Patricia Reynaud-Bouret. Reconstructing the functional connectivity of multiple spike trains using Hawkes models. *Journal of Neuroscience Methods*, 297:9–21, 2018.

[7] Emanuele Marconi, Thierry Nieus, Alessandro Maccione, Pierluigi Valente, Alessandro Simi, and others. Emergent Functional Properties of Neuronal Networks with Controlled Topology. *PLoS One*, 7(4):e34648, 2012.

[8] E. De Maria, A. Muzy, D. Gaffé, A. Ressouche, and F. Grammont. Verification of temporal properties of neuronal archetypes modeled as synchronous reactive systems. In Eugenio Cinquemani and Alexandre Donzé, editors, *Hybrid Systems Biology - 5th International Workshop, HSB 2016, Grenoble, France, October 20-21, 2016, Proceedings*, pages 97–112, 2016.

[9] Elisabetta De Maria, Thibaud L'Yvonnet, Daniel Gaffé, Annie Ressouche, and Franck Grammont. Modelling and formal verification of neuronal archetypes coupling. In *Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics, Nha Trang City, Viet Nam, December 7-8, 2017*, pages 3–10, 2017.

[10] Abdorrahim Bahrami, Elisabetta De Maria, and Amy Felty. Modelling and verifying dynamic properties of biological neural networks in coq. In *Proceedings of the 9th International Conference*

*on Computational Systems-Biology and Bioinformatics*, CSBio 2018, pages 12:1–12:11, New York, NY, USA, 2018. ACM.

[11] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: An Introduction.* Cambridge University Press, New York, NY, USA, 2002.

[12] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

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

[14] David R. Gilbert and Monika Heiner. Advances in computational methods in systems biology. *Theor. Comput. Sci.*, 599:2–3, 2015.

[15] François Fages, Sylvain Soliman, and Nathalie Chabrier-rivier. Modelling and querying interaction networks in the biochemical abstract machine biocham. *Journal of Biological Physics and Chemistry*, 4:64–73, 2004.

[16] Adrien Richard, Jean-Paul Comet, and Gilles Bernot. Graph-based modeling of biological regulatory networks: Introduction of singular states. In *Computational Methods in Systems Biology, International Conference, CMSB 2004, Paris, France, May 26-28, 2004, Revised Selected Papers*, pages 58–72, 2004.

[17] Elisabetta De Maria, François Fages, Aurélien Rizk, and Sylvain Soliman. Design, optimization and predictions of a coupled model of the cell cycle, circadian clock, DNA repair system, irinotecan metabolism and exposure control under temporal logic constraints. *Theor. Comput. Sci.*, 412(21):2108–2127, 2011.

[18] Carolyn L. Talcott and Merrill Knapp. Explaining response to drugs using pathway logic. In *Computational Methods in Systems Biology - 15th International Conference, CMSB 2017, Darmstadt, Germany, September 27-29, 2017, Proceedings*, pages 249–264, 2017.

[19] N. Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[20] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[21] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[22] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.

[23] Luke webpage. `http://www.it.uu.se/edu/course/homepage/pins/vt11/lustre`.

[24] A. Franzén. Using satisfiability modulo theories for inductive verification of lustre programs. *Electr. Notes Theor. Comput. Sci.*, 144(1):19–33, 2006.

[25] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9, Nov 2008.

[26] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 510–517, 2016.

[27] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[28] Mariem Abdelmoula, Daniel Gaffé, and Michel Auguin. Automatic test set generator with numeric constraints abstraction for embedded reactive systems: Autseg v2. In *SIMUL 2015: The Seventh International Conference on Advances in System Simulation*, SIMUL 2015: The Seventh International Conference on Advances in System Simulation, Barcelone, Spain, November 2015.

[29] Klaas Wijbrans, Franc Buve, Robin Rijkers, and Wouter Geurts. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. volume 5014, pages 419–424, 05 2008.

[30] Stacy D. Nelson and Charles Pecheur. Formal verification for a next-generation space shuttle. In *FAABS*, 2002.

[31] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[32] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[33] E. M. Clarke, D. E. Long, and K. L. Mcmillan. Compositional model checking. MIT Press, 1999.

[34] Pascal Fontaine, editor. *Proceedings of the 27th International Conference on Automated Deduction*, volume 11716 of *Lecture Notes in Computer Science*. Springer, August 2019.

[35] John Harrison, John O'Leary, and Andrew Tolmach, editors. *Proceedings of the 10th International Conference on Interactive Theorem Proving*, volume 141 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, September 2019.

[36] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[37] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[38] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[39] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[40] Xavier Leroy et al. The CompCert project: Compilers you can formally trust. `http://compcert.inria.fr/`.

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. ACM, 2011.

[42] Gerwin Klein et al. Sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207—220. ACM, 2009.

[43] Georges Gonthier et al. A machine-checked proof of the odd order theorem. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[44] Thomas Hales et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[45] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952.

[46] René Thomas, Denis Thieffry, and Marcelle Kaufman. Dynamical behaviour of biological regulatory networks—i. biological role of feedback loops and practical use of the concept of the loop-characteristic state. *Bulletin of Mathematical Biology*, 57(2):247–276, Mar 1995.

[47] Venkatramana N. Reddy, Michael L. Mavrovouniotis, and Michael N. Liebman. Petri net representations in metabolic pathways. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, USA, July 1993*, pages 328–336, 1993.

[48] Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Proceedings of the 6th Pacific Symposium on Biocomputing, PSB 2001, Hawaii, USA, January 3-7, 2001*, pages 459–470, 2001.

[49] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[50] Nathalie Chabrier-Rivier, Marc Chiaverini, Vincent Danos, François Fages, and Vincent Schächter. Modeling and querying biomolecular interaction networks. *Theor. Comput. Sci.*, 325(1):25–44, 2004.

[51] R. Hofestädt and S. Thelen. Quantitative modeling of biochemical networks. *In Silico Biology*, 1:39–53, 1998.

[52] Rajeev Alur, Calin Belta, and Franjo Ivancic. Hybrid modeling and simulation of biomolecular networks. In *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, pages 19–32, 2001.

[53] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology, International Conference, CMSB 2007, Edinburgh, Scotland, September 20-21, 2007, Proceedings*, pages 184–199, 2007.

[54] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

[55] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 495–499, 1999.

[56] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 585–591, 2011.

[57] Elisabetta De Maria, Joëlle Despeyroux, and Amy P. Felty. A logical framework for systems biology. In *Formal Methods in Macro-Biology - First International Conference, FMMB 2014, Nouméa, New Caledonia, September 22-24, 2014. Proceedings*, pages 136–155, 2014.

[58] M. Dénès, B. Lesage, Y. Bertot, and A. Richard. Formal proof of theorems on genetic regulatory networks. In *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 69–76, Sep. 2009.

[59] Adnan Rashid, Osman Hasan, Umair Siddique, and Sofiane Tahar. Formal reasoning about systems biology using theorem proving. *PLOS ONE*, 12(7):1–27, 07 2017.

[60] Bruno Pinaud, Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, and Jason Vallet. PORGY : a visual analytics platform for system modelling and analysis based on graph rewriting. In *17ème Journées Francophones Extraction et Gestion des Connaissances, EGC 2017, 24-27 Janvier 2017, Grenoble, France*, pages 473–476, 2017.

[61] W. Maas. Networks of spiking neurons: The third generation of neural network models. *Trans. Soc. Comput. Simul. Int.*, 14(4):1659–1671, December 1997.

[62] H. Paugam-Moisy and S. M. Bohte. Computing with spiking neuron networks. In *Handbook of Natural Computing*, pages 335–376. 2012.

[63] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[64] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004.

[65] L. Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *J Physiol Pathol Gen*, 9:620–635, 1907.

[66] Bogdan Aman and Gabriel Ciobanu. Modelling and verification of weighted spiking neural systems. *Theoretical Computer Science*, 623:92 – 102, 2016.

[67] Elisabetta De Maria and Cinzia 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, Funchal, Madeira, Portugal, January 19-21, 2018.*, pages 17–28, 2018.

[68] D. Purves, G. J. Augustine, D. Fitzpatrick, W. C. Hall, A.S. LaMantia, J. O. McNamara, and S. M. Williams, editors. *Neuroscience*. Sinauer Associates, Inc., 3rd edition, 2006.

[69] *Coq reference manual. Retrieved from https://coq.inria.fr/distrib/current/-refman/index.html.*

[70] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[71] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[72] E. De Maria, A. Muzy, D. Gaffé, A. Ressouche, and F. Grammont. Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models. Research Report 8937, UCA, Inria ; UCA, I3S ; UCA, LEAT ; UCA, LJAD, July 2016.

[73] Egidio D'Angelo, Giovanni Danese, Giordana Florimbi, Francesco Leporati, Alessandra Majani, Stefano Masoli, Sergio Solinas, and Emanuele Torti. The human brain project: High performance computing for brain cells hw/sw simulation and understanding. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 740–747, 2015.

[74] D. O. Hebb. Organization of behavior. new york: Wiley. *The Journal of Physiology*, 6(307):335, 1949.

[75] F. Grammont and A. Riehle. Spike synchronization and firing rate in a population of motor cortical neurons in relation to movement direction and reaction time. *Biological Cybernetics*, 88(5):360–373, May 2003.

**Elisabetta De Maria** is associate professor at Université Côte d'Azur (France). From 2011 to 2013 she was the coordinator of the International Research Master Program "Computational Biology and Biomedicine" (CBB) of University of Nice (France). Her expertise in bioinformatics led her to be the program chair of the conferences BIOINFORMATICS 2019 and 2020 (10th and 11th International Conference on Bioinformatics Models, Methods and Algorithms), and CSBio 2019 (10th International Conference on Computational Systems-Biology and Bioinformatics). Her research aims at substantiating the claim that formal methods of computer science can provide effective solutions to solve biological and medical problems.

**Abdorrahim Bahrami** is a PhD candidate in Computer Science at University of Ottawa since 2016, working under the supervision of Dr. Amy Felty (University of Ottawa, Ottawa, Canada) and Dr. Elisabetta De Maria (University of Nice, Nice, France). His thesis, entitled "Verified Model of Neurons, Basic Structures, and Neuronal Archetypes of Human Neural Networks: A Formal Methods Approach", involves analytical studies, implementation, and modelling of biological neural networks using formal verification. In 2009, he was awarded a bronze medal in SAT competition for the best program in crafted data category. He received another bronze medal and the best student solver award in SAT race 2010.

**Thibaud L'Yvonnet** started his studies in biology at Université Côte d'Azur (France). He is currently PhD student at INRIA Sophia Antipolis Méditerranée. His PhD research is supervised by Sabine Moisan (INRIA, Université Côte d'Azur) and Elisabetta De Maria (I3S, Université Côte d'Azur). Funded by Région Provence Alpe Côte d'Azur, his PhD thesis aims at analyzing cognitive impairment patient's behavior in situation of medical serious games using formal methods of computer science.

**Amy Felty** is a Professor of Computer Science at the University of Ottawa. She is an expert in formal methods and has over 25 years of experience in applying them in various domains. She is widely known for her work on logical frameworks and their application in domains such as programming languages, systems biology, privacy and security, and access control policy analysis. She is on the editorial board of several journals, including Springer Nature's Journal of Automated Reasoning. She has served as a Steering Committe Member, Program Chair or Co-Chair, and Program Committee Member of over 60 conferences and workshops in logic and formal methods. She is currently Treasurer of the ACM Special Interest Group on Logic and Computation.

**Daniel Gaffé** received the Ph.D. degree in electrical engineering in 1996 from the University of Nice-Sophia Antipolis. In 1997, he was recruited as an Associate Professor at the Université Nice Sophia-Antipolis, actual Université Côte d'Azur. He is doing his research at the LEAT (Laboratoire d'Electronique, Antennes et Telecommunications) since 2007. His studies concerns synchronous languages. In this framework, he develops the Light Esterel language with INRIA Institut. He applies the synchronous languages to the neural networks modelling too.

**Annie Ressouche** received her PhD degree in mathematics from the University of Paris 7 in 1978. Since 1979 she was a researcher at Inria, in Sophia Antipolis research center. First she contributed to the development of the Esterel language and the synchronous model of time and she studied model checking techniques for this model. In 2002, she joined the Stars team where she applies the synchronous modeling to the generation of safe activity recognition systems. In collaboration with the Côte d'Azur University she studied the safe composition of components in component-based adaptive reactive middleware and she develops the Light Esterel synchronous language. She applies the synchronous approach to neural networks modeling.

**Franck Grammont** is assistant professor in neurophysiology, theoretical neuroscience and cognitive sciences at the Dieudonné Laboratory of the Côte d'Azur University (Nice, France). His main interests concern the theory of neuronal archetypes, the theory of neuronal assemblies, the analysis of neuronal activity and the study of sensorimotor systems.