Modelling and Verifying Dynamic Properties of Biological Neural Networks in Coq

Abdorrahim Bahrami
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
abahr010@uottawa.ca

Elisabetta De Maria
Université Côte d'Azur
CNRS, I3S
France
elisabetta.de-maria@unice.fr

Amy Felty
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
afelty@uottawa.ca

ABSTRACT

Formal verification has become increasingly important because of the kinds of guarantees that it can provide for software systems. Verification of models of biological and medical systems is a promising application of formal verification. Human neural networks have recently been emulated and studied as a biological system. Some recent research has been done on modelling some crucial neuronal circuits and using model checking techniques to verify their temporal properties. In large case studies, model checkers often cannot prove the given property at the desired level of generality. In this paper, we provide a model using the Coq Proof Assistant and prove properties concerning the dynamic behavior of some basic neuronal structures. Understanding the behavior of these modules is crucial because they constitute the elementary building blocks of bigger neuronal circuits. By using a proof assistant, we guarantee that the properties are true for any input values, any length of input, and any amount of time. With such a model, there is the potential to detect inactive regions of the human brain and to treat mental disorders. Furthermore, our approach can be generalized to the verification of other kinds of networks, such as regulatory, metabolic, or environmental networks.

CCS CONCEPTS

• Applied Computing~Systems Biology • Theory of Computation~Logic and Verification

KEYWORDS

Biological network reconstruction and analysis, Modelling and simulation of biological processes and pathways, Human Neural Networks, Dynamic Properties, Leaky Integrate and Fire Model, Neuronal Modules, Archetypes, Formal Verification, Theorem Proving, Coq Proof Assistant

CSBio 2018, December 10–13, 2018, Bangkok, Thailand

ACM ISBN 978-1-4503-6560-4/18/12…$15.00
https://doi.org/10.1145/3291757.3291771

1 Introduction

In this work, we apply formal verification to verify the dynamic behavior of biological human neural networks. We focus on theorem proving, which can be used to show that a piece of software or a system is free of errors with respect to a formal model that is provided for it. In formal verification, often a model of the system is defined based on a transition graph [1]. Each node in the graph represents a state of the system being modelled and each edge stands for a transition from a source to a destination state. Model checkers or theorem provers 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 [2]. A variety of biological systems can be modelled as graphs whose nodes represent the different possible configurations of a 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 [3, 4]. 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 [5, 6]. By understanding and proving properties of a biological system, there is a higher chance of treating diseases and developing medicines that will be suited for them. Furthermore, weak points of the system can be detected and better prevention against disease and other external problems can be proposed. Finally, biological system recovery after damage has occurred can be studied and verified. In summary, behavior, disease, effects of medicine, external problems, environmental change impacts, and system recovery of a biological system can all be detected and verified using formal verification.

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 [7], Petri nets [8], π-calculus [9], bio-ambients [10], and reaction rules [11]. To capture the dynamics from a quantitative point of view, ordinary or stochastic differential equations are used extensively. More recent approaches include hybrid Petri nets [12] and hybrid automata [13],

stochastic π-calculus [14], and rule-based languages with continuous/stochastic dynamics such as Kappa [15]. 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 [16] or PRISM [17].

In [18], 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 [19], 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 [20]. It is a visual environment which allows modelling of biochemical systems as rule-based models. Rewriting strategies are used to choose the rules to be applied.

As far as human neural networks are concerned, there is recent work that has focused on their formal verification. In [21, 22], the authors consider the synchronous paradigm to model and verify some specific graphs composed of a few biological neurons. These graphs or mini-circuits, characterized by biologically relevant structures and behaviors, are referred to as archetypes and constitute the fundamental elements of neuronal information processing. They can be coupled to create the elementary building blocks of bigger neuronal circuits. For this reason, their study has become an emerging question in the domain of neurosciences, especially for their potential integration with neurocomputational techniques [23]. Furthermore, understanding these micro-circuits can help in detecting weakly active or inactive zones of the human brain, and in identifying neurons whose role is crucial to perform some vital activities, such as breathing or moving.

In the work proposed in [21, 22], some model checkers such as Kind2 [24] are employed to automatically verify properties concerning the dynamics of six basic archetypes and their coupling. However, model checkers prove properties for some given parameter intervals, and do not handle inputs of arbitrary length. In our work, we use the Coq Proof Assistant [25] to prove four important properties of neurons and archetypes. Coq implements a highly expressive higher-order logic in which we can directly introduce datatypes modelling neurons and archetypes, and express properties about them. As a matter of fact, one of the main advantages of using Coq is the generality of its proofs. Using such a system, we can prove properties about arbitrary values of parameters, such as any length of time, any input sequence, or any number of neurons. We use Coq's general facilities for structural induction and case analysis, as well as Coq's standard libraries that help in reasoning about rational numbers and functions on them. We believe the approach introduced in this paper for reasoning about neural networks is very promising, because it can be exploited for the verification of other kinds of biological networks, such as gene regulatory, metabolic, or environmental networks.

The paper is organized as follows. In Section 2, we introduce the state of the art relative to neural network modelling and the application of formal methods in this domain. In Section 3, we describe the computational model we have chosen, the Leaky Integrate and Fire model (LI&F), and we briefly introduce some basic archetypes. In Section 4, we introduce the Coq Proof Assistant. In Section 5, we present our model of neural networks in Coq, which includes definitions of neurons, operations on them, and combining them into archetypes. In Section 6, we present and discuss four important properties, starting with properties of single neurons and the relation between the input and output, and moving toward more complex properties that express their interactions and behaviors as a system. We provide a full proof of the first property; the remaining proofs are found in the appendix at the end of this paper. Finally, in Section 7, we conclude and discuss future work. The accompanying Coq code can be found at: http://www.site.uottawa.ca/~afelty/csbio18/.

2 Background

Neurons are the smallest unit of a neural network [26]. They are basically just a single cell. We can consider them simply as a function with one or more inputs and a single output. A human neuron receives its inputs via its dendrites. Dendrites are short extensions connected to the neuron body, which is called a soma. Inputs are provided in the form of electrical pulses (spikes). For each neuron there is another extension, called the axon, which plays the role of output. This extension is also connected to the cell body, but it is longer than the dendrites. Each neuron has its own activation threshold which is coded somehow inside the soma. When the sum of the spikes received by a neuron through its dendrites passes its threshold, the neuron fires a spike in the axon. Neurons can be connected to other neurons. Connections happen between the axon of a neuron and a dendrite of another neuron. Theses connections are called synaptic connections and the location of the connection is called a synapse. They are responsible for transmitting signals between neurons.

In this paper, we consider third generation models of neural networks. They are known as spiking neural networks [27] and have been proposed in the literature with different complexities and capabilities. In this work we focus on the Leaky Integrate and Fire (LI&F) model originally proposed in [28]. It is a computationally eŎcient approximation of a single-compartment model [29] and is abstract enough to be able to apply formal verification techniques. In such a model, neurons integrate present and past inputs in order to update their membrane potential values. Whenever the potential exceeds a given threshold, an output signal is fired.

As far as spiking neural networks are concerned, in the literature there are a few attempts at giving formal models for them. In [30], 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 [31] to model LI&F networks. This modelling is substantially different from the one proposed in [30] 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.

Another recent application of formal methods in computer science to neuro-sciences is given in [21, 22], where the authors model LI&F neurons and some basic small circuits using the synchronous language Lustre. Such a language is dedicated to the modelling of reactive systems, i.e., systems which constantly interact with the environment and which may have an infinite duration. It relies on the notion of logical time: time is considered as a sequence of discrete instants, and an instant is a point in time where external input events can be observed, computations can be done, and outputs can be emitted. Lustre is used not only to encode neurons and some basic archetypes (simple series, parallel composition, etc.), but also some properties concerning their dynamic evolution. Some model checkers are then employed to automatically prove these properties for some given parameter intervals.

LI&F networks extended with probabilities are formalized as discrete-time Markov chains in [32]. The proposed framework is then exploited to propose an algorithm which reduces the number of neurons and synaptic connections of input networks while preserving their dynamics.

3 Leaky Integrate and Fire Model and Neuron Modules

Leaky Integrate and Fire (LI&F) networks [33] can be seen as directed weighted graphs whose nodes stand for neurons and whose edges represent synaptic connections. The signals propagating over synapses are trains of impulses and they are referred to as *spikes*. Synapses may modulate these signals according to their weight: *excitatory* if positive, or *inhibitory* if negative.

The dynamic behavior of neurons is guided by their *membrane potential* (or, simply, potential), which represents the difference of electrical potential across the cell membrane. The membrane potential of each neuron depends on the spikes received over its input synapses. Both current and past spikes are taken into account, but the contribution of old spikes is less important. In particular, the *leak factor* is introduced to weaken the signals received in the past. The neuron outcome depends on the difference between its membrane potential and its *firing threshold*: it is enabled to fire (i.e., emit a spike over all its outgoing synapses) only whenever such a difference is positive. Immediately after each spike emission, the neuron membrane potential is reset to zero.

More formally, the following definition can be given for networks of LI&F neurons.

Definition 1 (LI&F Neural Network). A LI&F Neural Network *is a tuple* (V, E, w)*, where:*

- V *is the set of LI&F neurons*,
- $E \subseteq V \times V$ *is the set of synapses*,
- $w: E \to \mathbb{Q} \cap [-1,1]$ *is the synapse weight function associating a weight* $w_{u,v}$ *to each synapse* (u, v).

We distinguish three disjoint sets of neurons: V_i *(input neurons),* V_{int} *(intermediary neurons), and* V_o *(output neurons), with* $V = V_i \cup V_{int} \cup V_o$.

A LI&F neuron *is characterized by a tuple* (τ, r, p, y)*, where:*

- $\tau \in \mathbb{Q}^+$ *is the* firing threshold *or* activation 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:*

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

with $p(0) = 0$ *and where* $x_i(t) \in \{0, 1\}$ *is the signal received at the time* t *by the neuron through its* i^{th}*out of* m *input synapses (observe that the past potential is multiplied by the leak factor while current inputs are not weakened),*

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

$$y(t) = \begin{cases} 1 & if\ p(t) \geq \tau \\ 0 & if\ p(t) < \tau \end{cases} \quad (2)$$

The set of neurons of a LI&F neural network can be divided into input, intermediary, and output neurons. Each input neuron can only receive external signals as input and the output of each output neuron is considered as an output for the network. Output neurons are the only ones whose output is not connected to other neurons.

In neural networks, it is possible to identify some mini-circuits with a relevant topological structure. Each one of these small modules, which are often referred to as *archetypes* in the literature, displays a specific class of behaviors. They can be coupled together to form the elementary bricks of bigger neural circuits. In [21], six basic archetypes have been identified and validated against some temporal logic properties thanks to model checking techniques [1]. They are the following ones (see Figure 1 for a graphical representation): (a) *simple series*, which is a sequence of neurons where each element of the chain receives as input the output of the preceding one; (b) *series with multiple outputs,* which 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); (c) *parallel composition*, which is a set of neurons receiving as input the output of a given neuron; (d) *negative loop*, which is a loop consisting of two neurons—the first neuron activates the second one while the latter inhibits the former; (e) *inhibition of a behavior*, which consists of two neurons, the first one inhibiting the second one; and (f) *contralateral inhibition*, which is made by two or more neurons, each one inhibiting the other ones. In Figure 1, a solid black circle at the end of an edge shows an inhibitory connection and a regular arrow represents an excitatory connection.

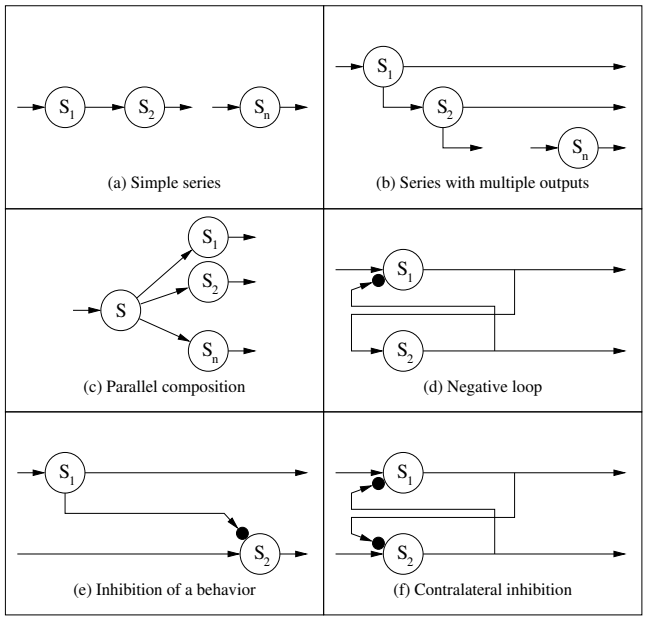

Figure 1: Neuron archetypes [21, 22]

In this paper, we exploit Coq to prove more general properties concerning some of these archetypes.

4 The Coq Proof Assistant

In this section, we present the basic elements of Coq that we use to represent our model. More complete documentation of Coq can be found in [25, 34]. Coq is a proof assistant that implements the Calculus of Inductive Constructions [35], which is an expressive higher-order logic. Using this software, we can express and prove properties in this logic. Expressions in the logic include a functional programming language. It is a typed language, which means that every Coq expression has a type. For instance, `X:nat` expresses that variable `X` is in the domain of natural numbers. The types used in our model include `nat`, `Q`, and `list` which denote natural numbers, rational numbers, and list of elements respectively. These types are found in Coq's standard libraries. Elements of a list have their own type. For instance, `L:list nat` means that `L` is a list of natural numbers. A list can be empty, which is written `[]` or `nil` in Coq. Functions are a basic element of any functional programming language. The general form of a function in Coq is shown in Figure 2.

```
Definition/Fixpoint Function_Name
(Input1: Type of Input1) …
(Inputn: Type of Inputn) : Output Type :=
  Body of the function.
```

Figure 2. General form for defining a function in Coq

`Definition` and `Fixpoint` are Coq keywords for defining non-recursive and recursive functions, respectively. Calling a function inside its body causes an error when `Definition` is used. After either one of these keywords comes the name that a programmer gives to the function. Following the function name are the input arguments and their types. If two or more inputs have the same type, they can be grouped as, for example, `(X Y Z: Q)` which means all variables `X`, `Y`, and `Z` are rational numbers. Following the inputs is a colon, followed by the type of the function. Finally, the body of the function is a Coq expression representing a program, followed by a dot.

Pattern matching is a useful method in Coq, used for case analysis. This feature is used, for instance, for distinguishing between base cases and recursive cases in recursive functions. For example, it can distinguish when a list is empty or not. The pattern for a non-empty list shows the first element of the list, which is called the *head*, followed by a double colon, followed by the rest of the list, which is called the *tail*. The tail of a list itself is a list of elements of the same type as the type of the head. For example, let *L* be the list `(6::3::8::nil)` containing three natural numbers. An alternate notation for Coq lists allows *L* to be written as `[6;3;8]` where the head is `6` and the tail is `[3;8]`. Thus, the general pattern for non-empty lists often used in Coq recursive functions has the form `(h::t)`. Another example of a Coq data type is the 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 example, 1 is represented as `(S 0)`, 2 as `(S (S 0))`, etc. In Figure 3, some patterns for lists and natural numbers are shown using Coq's `match…with…end` pattern matching construct.

```
match X with
| 0 => Do something when X = 0
| S n => Do something when X is successor of n
end

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

Figure 3. General form for pattern matching of natural numbers and lists in Coq

In addition to the data types that are defined in Coq's libraries, new data types can be defined. One way to do so is using records. Records can have different fields with different types. For example, we can define a record that has 3 fields `Fieldnat`, `FieldQ`, and `ListField`, which have types natural number, rational number, and list of natural numbers, respectively. Figure 4 shows the Coq syntax for the definition of this record with one additional field called `CR`.

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

S: Sample_Record
```

Figure 4. Definition of a record and a variable with the record type in Coq

Fields in Coq can represent conditions on other fields. For example, field `CR` in Figure 4 is a condition on the `Fieldnat` field stating that it must be greater than 7. After defining a record, it is a type like any other type, and so for example, we can have variables with the new record type. Variable `S` shown with type `Sample_Record` in Figure 4 is an example.

5 Modelling Human Neural Networks in Coq

```
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.
```

Figure 5. Coq code defining a neuron and the weighted sum of its inputs

We illustrate our encoding of neural networks in Coq by beginning with the code in Figure 5. We use Coq's record structure to define a neuron. This record includes five fields with their types, and four fields which represent constraints that the first five fields must satisfy according to the LI&F model mentioned in Section 3. The types include natural numbers, rational numbers, and lists. In particular, a neuron's output (`Output`) is represented as a list of natural numbers, with one entry for each time step. The weights attached to the inputs of the neuron (`Weights`) are stored in a list of rational numbers, one for each input in some designated order. The leak factor (`Leak_Factor`), the firing threshold (`Tau`), and the most recent neuron membrane potential (`Current`) are rational numbers. With respect to the four conditions, for example, consider `PosTau`, which expresses that `Tau` must be positive. `Qle_bool` and other arithmetic operators come from Coq's rational number library. The other three 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 range of [-1, 1]. We omit the definitions of `Bin_List` and `WeightInRange` used in these statements. The reader is referred to the accompanying Coq code.

Given a neuron `N`, we write `(Output N)` to denote its first field, and similarly for the others. To create a new neuron with values `O`, `W`, `L`, `T`, and `C` of the appropriate 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 Figure 5 implements the weighted sum of the inputs of a neuron, which is an important part of the calculation in Equation (1). In this recursive function, there are two arguments: `Weights` representing $w_1, \ldots, w_m$ and `Inputs` representing $x_1, \ldots, x_m$. The function returns an element of type `Q`. Its definition uses pattern matching on both inputs simultaneously. The body of the definition uses Booleans, the `if` statement, and the equality operator on natural numbers (`beq_nat`), all from Coq's standard library. Natural numbers, such as `0%nat` above are marked with their type to distinguish them from rational numbers, whose types are omitted. Although, we always call the `potential` function with two lists of equal length, Coq requires functions to be total; when two lists do not have equal length, we return a "default" value of 0. Also, when we call this function, `Inputs`, which is the second argument of the function, will always be a binary list (contains only the natural numbers 0 and 1). Thus, when head of this list `h2` is 0, we don't need to add anything to the final sum because anything multiplied by 0 is 0. In this case, we just call the function recursively on the remaining weights and inputs `t1` and `t2`, respectively. On the other hand, when `h2` is 1, we need to add `h1`, the head of `Weights` to the final sum, which again is the recursive call on `t1` and `t2`.

```
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).
```

Figure 6. Coq code defining neuron potential function

Figure 6 shows the `NextPotential` function, which implements $p(t)$ from Equation (1). Recall that (`Current N`) is the most recent potential value of the neuron which is $p(t-1)$ in Equation (1). `(Qle_bool (Tau N) (Current N))` represents $\tau \leq p(t-1)$ and we use the potential function defined in Figure 5 for the part calculating the weighted sum of the neuron inputs. Finally, the last line implements $r \cdot p(t-1)$.

Figure 7 on the next page contains two definitions. The first calculates the next output of the neuron which is $y(t)$ in Equation (2). Recall that `(NextPotential N Inputs)` shown in Figure 6 calculates $p(t)$. Thus, the expression `(Qle_bool (Tau N) (NextPotential N Inputs))` expresses the condition $\tau \leq p(t)$.

In our model, the state of a neuron is represented by the `Output` and `Current` fields. The `Output` field of a neuron in the initial state is `[0%nat]`, which denotes a list of length 1 containing only 0. The `Current` field represents the initial potential, which is set to 0. A neuron changes state by processing input. After processing a list of n inputs, the `Output` field will be a list of length $n+1$ containing 0's and 1's, and the `Current` field will be set to the value of the potential after processing these

n inputs. State change occurs by applying the `NextNeuron` function in Figure 7 to a neuron and a list of inputs. As is typical in functional programming, we represent a neuron at its later state by creating 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 simplifies proofs by induction over lists, which we use regularly in our Coq proofs. Thus, the most recent output of the neuron is at the of head the list. We can see this in the code in Figure 7, where the new value of the output is `((NextOutput N Inputs)::(Output N))`. The next output of the neuron is at the head, 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 potential value of the neuron or $p(t-1)$. So, for calculating the next potential value of the neuron or $p(t)$, the `NextPotential` function in Figure 6 is called.

Following the new values for each field of the neuron, we have proofs of the four constraints. The first requires a lemma `NextOutput_Bin_List` (statement omitted) which allows us to prove 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 are about components of the neuron that do not change.

```
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).
```

Figure 7. Coq functions for returning the next output and neuron structure after the next time step

To reinitialize a neuron to the initial state as described above, the `ResetNeuron` function is used. This function takes any `Neuron` as input, and returns a new one, with the `Output`, `Current`, and `Output_Bin` fields reset, while keeping the others.

So far, we have discussed the encoding and processing of single neurons in isolation, which take in inputs and produce outputs. The archetypes in Figure 1 illustrate some ways in which networks of neurons are connected. We have so far considered archetypes (a) and (e) in our work. We represent (a) as an ordered list of single input neurons (a Coq list of type `list Neuron`), where we assume that the output of a neuron in the list is connected to the input of the neuron occurring immediately following it, the input to the first is the input to the whole series, and the output of the last is the output of the whole series. To represent (e), we define Coq predicates that relate two neurons; an example is discussed further in Section 6.

6 Properties of Neural Networks and their Proofs

As mentioned earlier, we prove four basic properties of the LI&F model of neurons in this section. All of them have been fully verified in Coq. We start in the next section with a property about a simple neuron, which has only one input. We refer to this neuron as a *single-input neuron*.

In all of the statements of the properties, we omit the assumption that the input sequence of the neuron is a binary list and contains only 0s and 1s. It is, of course, included in the Coq code. We use several other conventions to enhance readability when stating properties and presenting proofs. For example, we state our 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 as $\tau(N)$, `(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. Also, we use notation and operators from the Coq standard library for lists. For instance, $length$ and $+$ are list operators; the former is for finding the number of elements in the list and the latter is the notation we will use here for list concatenation.

In addition, although for a neuron N, the list $Output(N)$ is encoded in reverse order in our Coq model, when presenting properties and their proofs here, we use forward order.

6.1 The Delayer Effect for a Single-Input Neuron

The first property is called the *delayer effect* property. Recall that a neuron is in an inactive state initially, which means the output of a neuron at time 0 is 0. When a neuron has only one input, and the weight of that input is greater than or equal to its activation threshold, then the neuron transfers the input sequence to the output without any change (except for a "delay" of length 1). For instance, if a single input neuron receives 0100110101 as its input sequence, it will produce 00100110101 as output. Neurons that have this property are not functional neurons. They are mainly just transferring signals. Humans have some of this type of neuron in their auditory system. This property is expressed as Property 1.

Property 1. $\forall\ (N: neuron)\ (input: list\ nat)$,

$$length\big(w(N)\big) = 1 \wedge w_1(N) \geq \tau(N) \rightarrow$$

$$Output(N') = [0] + input$$

In the above statement, N' denotes the neuron obtained by initializing N and then processing the input (using $ResetNeuron$ and repeated applications of $NextNeuron$). We use this convention in stating all of our properties. Note that in Definition 1, Equation (1), p is a function of time. Time in our Coq model is encoded as the position in the output list. If $Output(N)$ has

length t, then $p(N)$ stores $p(t-1)$ from Equation (1). If we then apply $NextNeuron$ to N and the next input obtaining N', then $Output(N')$ has length $t+1$ and $p(N')$ stores the value $p(t)$ from Equation (1).

In order to prove Property 1, we need the following lemma, which states that when a neuron has one input and its input weight is greater than or equal to its threshold, the potential value of that neuron is always non-negative.

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

As explained above $p(N')$ is the most recent value of the potential function of neuron N, i.e., the one obtained after processing all of the input values. The proof of this lemmas is in the appendix. We use it here to prove Property 1.

Proof *(of Property 1).* 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 $(input + [\square])$ for some additional input value $\square$. 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 $Output\big(N''\big) = [0] + input'$.

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 $\square = 0$ or $\square = 1$. We break this into two different cases, depending on the value of $\square$.

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 Equation (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 Equation (1), 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 Equation (2) in Definition 1, the next output of the neuron will be 0. The other case, which comes from the second clause of Equation (1) 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 Equation (2), 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'' using Equation (1):

$p(N'') = w_1(N') \cdot 1 = w_1(N')$ 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 Equation (2), 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 Equation (2), 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.

6.2 The Filter Effect for a Single Neuron

The next property we consider is also about single-input neurons. When a neuron has only one input, and the weight of that input is less than its activation threshold, the neuron passes on the value 1 once as output for each sequence of n 1s in the input, where n is the designated length of the series. All 1s in the input are replaced by 0 except for the $n^{t\square}$ one, the $2n^{t\square}$ one, the $3n^{t\square}$ one, etc. The other 1s are "filtered out." For instance, let $n = 3$. Then if a single input neuron with this effect receives 01110010101 as input, it will produce 000010000001 as the output sequence. (The output sequence is one longer than the input because of the leading 0.) As a consequence, there are never two consecutive 1s in the output sequence. This consequence is called the *filter effect*. Most neurons in a human body have the filter effect because their input weight is less than their activation threshold. Normally, more than one input is needed to activate a human neuron. In biology, this property is often called the *integrator effect.*

Property 2. $\forall\ (N: neuron)\ (input: list\ nat)$,
$length\big(w(N)\big) = 1 \wedge w_1(N) < \tau(N) \rightarrow 11\ \notin Output(N')$

Note that in the statement above, $11 \notin Output(N')$ means there are no two consecutive 1s in the list $Output(N')$. This theorem is also proved by induction on the structure of the input list. (See the appendix.)

6.3 The Inhibitor Effect

The next property is an important one because it has the potential to help us detect inactive zones of the brain. Normally, a human neuron does not have negative weights for all of its inputs but when one or more positive weight inputs are out of order because of some kind of disability, this property can occur. It is called the *inhibitor effect* because it is important for proving properties of archetype 1(e). We consider here the single neuron case. When a neuron has only one input and the weight of that input is less than 0, then the neuron is inactive, which means that for any input, the neuron cannot emit 1 as output. i.e., if a signal reaches this neuron, it will not pass through. As with the other properties, the input sequence has an arbitrary finite length. This property is expressed as follows.

Property 3. $\forall\ (N: neuron)\ (input: list\ nat)$,
$length\big(w(N)\big) = 1 \wedge w_1(N) < 0 \rightarrow 1\ \notin Output(N')$

Similarly, in the statement above, $1 \notin Output(N')$ means there is no 1 in the list $Output(N')$. This property is also proved by induction on the structure of the input, again in the appendix.

The inhibitor effect expressed in Property 3 has a more general version, which we plan to prove as a future work. For a neuron with multiple inputs, when all input weights are less than or equal to 0, then the neuron is inactive and can't pass any signal. Thus, in addition to proving this property for arbitrary input length, we intend to generalize it to an arbitrary number of neurons. As mentioned, recognizing inactive neurons can help to detect inactive zones of the brain. In addition, it can also help to simplify the structure of a neural network by removing such neurons from the network.

6.4 The Delayer Effect in a Series of Single Neurons

The next property is about the archetype shown in Figure 1(a). In this structure, each neuron output is the input of the next neuron. 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. This property is expressed as follows.

Property 4. $\forall\ (Series: list\ neuron)\ (input: list\ nat)\ (i: nat)$,
$length(Series) = n \wedge 0 \leq i < n \wedge$
$length(w(Series[i]) = 1 \wedge w_1(Series[i]) > \tau(Series[i])$
$\rightarrow Output = zeros(n) + input$

This time, the proof proceeds by induction on the length of $Series$. As with the other properties, the complete proof is in the appendix.

7 Conclusion

In this work, we proposed a formal approach to model and validate leaky integrate and fire neurons and some basic circuits (simple series and inhibition of a behavior). In the literature, this is not the first attempt to the formal investigation of neural networks. In [21, 22], the synchronous paradigm has been exploited to model neurons and some small neuronal circuits with a relevant topological structure and behavior and to prove some properties concerning their dynamics. Our approach based on the use of the Coq proof assistant (which is, to the best of our knowledge, the first one), turned out to be much more general. As a matter of fact, we guarantee 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. In [22], the authors were able to write a function (more precisely, a Lustre node) which encodes the expected behavior of the circuit. Then, they could call a model checker to test whether the property at issue is valid for some input series with a fixed length. Here we can prove that the wished behavior is true whatever the length and the parameters of the series are.

As a first next step, we intend to formally study the missing archetypes of Figure 1 (series with multiple outputs, parallel composition, negative loop, and contralateral inhibition) and other new archetypes made of two, three or more neurons. We already started to investigate the two-neuron positive loop, where the first neuron activates the second one, which in turn activates the first one. Our progress so far includes defining a Coq inductive predicate that relates these two neurons and their corresponding two lists of values obtained by applying the potential function over time. This predicate is true whenever the output has a particular pattern that is important for proving one of the more advanced properties we are studying. Defining general relations that can be specialized to specific patterns will likely also be very useful for the kinds of properties that are important for more complex networks.

As a second next step, we plan to focus on the composition of the studied archetypes. There are two main ways to couple two circuits: either to connect the output of the first one to the input of the second one, or to nest the first one inside the second one. We are interested in detecting the compositions which lead to circuits with a meaningful biological behavior. Archetypes can be considered as the syllables of a given alphabet. When two or more syllables are combined, it is possible to obtain either a real word or a word which does not exist. At the same way, the archetype composition can lead to meaningful networks or not.

As a long-term aim, we would like to be able to prove that whatever neural network can be expressed as a combination of the small mini-circuits we have identified, as far as all the words can be expressed as combination of the syllables of a given alphabet. 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.

ACKNOWLEDGMENTS

The first and third authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada. We thank neurophysiologist Frank Grammont for his useful explanations of neuron functioning.

REFERENCES

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

[2] D.R. Gilbert and M. Heiner. 2015. Advances in computational methods in systems biology. *Theoretical Computer Science*, 599, 2-3.

[3] F. Fages, S. Soliman, and N. Chabrier-Rivier. (2004). Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM. *Journal of Biological Physics and Chemistry*, 4(2), 64–73.

[4] A. Richard, J.P. Comet, and G. Bernot. 2004. Graph-based modeling of biological regulatory networks: Introduction of singular states. In *International Conference on Computational Methods in Systems Biology (CMSB '04)*, pp. 58-72.

[5] E. De Maria, F. Fages, A. Rizk, and S. Soliman. 2011. 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. *Theoretical Computer Science* 412(21), 2108-2127.

[6] C.L Talcott, and M. Knapp. 2017. Explaining response to drugs using pathway logic. In *International Conference on Computational Methods in Systems Biology (CMSB '17)*, pp. 249-264.

[7] R. Thomas, D. Thieffry, and M. Kaufman. 1995. 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.

[8] V.N Reddy, M.L. Mavrovouniotis, and M.N. Liebman. 1993. Petri net representations in metabolic pathways. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology (ISMB '93)*. pp. 328-336. AAAI Press.

[9] A. Regev, W. Silverman, and E.Y. Shapiro. 2001. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Proceedings of the sixth Pacific Symposium of Biocomputing (PSB '01),* pp. 459-470.

[10] A. Regev, E.M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. (2004). Bioambients: An abstraction for biological compartments. *Theoretical Computer Science* 325(1), 141-167.

[11] N. Chabrier-Rivier, M. Chiaverini, V. Danos, F. Fages, and V. Schächter. (2004). Modeling and querying biochemical interaction networks. *Theoretical Computer Science* 325(1), 25-44.

[12] R. Hofestädt and S. Thelen. 1998. Quantitative modeling of biochemical networks. In *Silico Biology*, vol. 1, pp. 39-53. IOS Press.

[13] R. Alur, C. Belta, F. Ivanicic, V. Kumar, M. Mintz, G.J. Pappas, H. Rubin, and J. Schug. 2001. Hybrid modeling and simulation of biomolecular networks. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control (HSCC'01),* Springer LNCS, vol. 2034.

[14] A. Phillips and L. Cardelli, L. 2004. A correct abstract machine for the stochastic pi-calculus. In *Proceedings of BioConcur*, Electronic Notes in Computer Science.

[15] V. Danos and C. Laneve. 2004. Formal molecular biology, *Theoretical Computer Science* 325(1), 69-110.

[16] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. 1999. NUSMV: A new symbolic model verifier. In *Proceedings of the 11th Intl. Conference on Computer Aided Verification.* pp. 495-499. CAV '99, Springer-Verlag, London, UK.

[17] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0 2001 Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of Springer LNCS, pp. 585-591.

[18] E. De Maria, J. Despeyroux, and A.P. Felty. 2014. A Logical Framework for Systems Biology, In *1st International Conference on Formal Methods in Macro-Biology (FMMB '14)*, Springer LNCS 8738, pp. 136-155.

[19] A. Rashid, O. Hasan, U. Siddique, and S. Tahar. 2017. Formal reasoning about systems biology using theorem proving. *PLoS ONE* 12(7): e0180179.

[20] O. Andrei, M. Fernández, H. Kirchner, and B. Pinaud. 2016. *Strategy-Driven Exploration for Rule-Based Models of Biochemical Systems with Porgy*. Research Report: Université de Bordeaux, Inria; King's College London, University of Glasgow.

[21] E. De Maria, A. Muzy, D. Gaffé, A. Ressouche, and F. Grammont. 2016. Verification of Temporal Properties of Neuronal Archetypes Modeled as Synchronous Reactive Systems. In *Hybrid Systems Biology - 5th International Workshop, (HSB '16),* Grenoble, France, October 20-21, 2016, pp. 97–112.

[22] E. De Maria, T. L'Yvonnet, D. Gaffé, A. Ressouche, and F. Grammont. 2017. Modelling and Formal Verification of Neuronal Archetypes Coupling. In *8th International Conference on Computational Systems-Biology and Bioinformatics (CSBio 2017)*, pp. 3-10.

[23] H. Markram. 2006. The blue brain project. *Nat Rev Neurosci* 7(2), 153–160.

[24] G. Hagen and C. Tinelli. 2008. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD '08)*, pp. 1–9.

[25] Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer.

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

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

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

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

[30] B. Aman and G. Ciobanu. 2016. Modelling and verification of weighted spiking neural systems. *Theoretical Computer Science* 623, 92 -102.

[31] E. De Maria and C. Di Giusto. 2018. Parameter Learning for Spiking Neural Networks Modelled as Timed Automata. In *9th International Conference on Bioinformatics Models, Methods and Algorithms (BIOINFORMATICS '18),* pp. 17-28.

[32] E. De Maria, D. Gaffé, C. Girard Riboulleau, and A. Ressouche. 2018. A Model-checking Approach to Reduce Spiking Neural Networks. In *9th International Conference on Bioinformatics Models, Methods and Algorithms (BIOINFORMATICS '18),* pp. 89-96.

[33] W. Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671.

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

[35] T. Coquand and G. Huet. 1988. The calculus of constructions. *Information and Computation*, 76, 95-120.

Appendix

We include here the complete proofs of Lemma 1 and Properties 2, 3, and 4.

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

As explained above $p(N')$ is the most recent value of the potential function of neuron N, i.e., the one obtained after processing all of input values.

Proof *(of Lemma 1)*. The proof is by induction on the length of the input sequence.

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, $p(N') = 0$. Therefore, $p(N') \geq 0$.

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 input after processing $input'$. By the induction hypothesis, we know $p(N') \geq 0$ and we must prove that $p(N'') \geq 0$.

Note that $\tau(N) = \tau(N') = \tau(N'')$ and $w_1(N) = w_1(N') = w_1(N'')$, so we use them interchangeably. We break this proof into two cases depending on whether or not $p(N') \geq \tau(N')$.

First, let's assume that $p(N') \geq \tau(N')$. Because the input contains only 0s and 1s, we know that $h = 0$ or $h = 1$. We calculate $p(N'')$, which as stated, corresponds to $p(t)$ in Equation

(1), i.e., the potential value of the neuron at time t; the value $p(N')$ represents $p(t-1)$ in this definition. Because of our assumption, only the first clause of Equation (1) applies, with two possibilities depending on the value of h:

$p(N'') = w_1(N') \cdot 0 = 0$ or $p(N'') = w_1(N') \cdot 1 = w_1(N')$.

In the first case, $p(N'') = 0 \geq 0$. In the second case, $p(N'') = w_1(N')$ and we know $w_1(N') \geq \tau(N)$ by assumption, and $\tau(N) > 0$ because, by definition, the activation threshold of any neuron is a positive value.

Second, we assume that $p(N') < \tau(N)$. So, again because the input contains only 0s and 1s, we know that $h = 0$ or $h = 1$. By the second clause of the definition of p, we have:

$p(N'') = r(N') \cdot p(N')$ or $p(N'') = w_1(N') + r(N') \cdot p(N')$.

In the first case, $r(N')$ is non-negative by definition and $p(N')$ is non-negative by the induction hypothesis. Thus, $r(N') \cdot p(N') \geq 0$. For the second case, we also have that $w_1(N') \geq \tau(N') > 0$, and thus the sum of two non-negative numbers is also non-negative.

This completes the proof, thus showing that it is always the case that the value of the potential of a single input neuron with a non-negative input weight will be non-negative.

Property 2. $\forall\ (N{:}\,neuron)\ (input{:}\,list\ nat)$,

$length\big(w(N)\big) = 1 \wedge w_1(N) < \tau(N) \rightarrow 11\ \notin Output(N')$

Note that in the statement above, $11 \notin Output(N')$ means there are no two consecutive 1s in the list $Output(N')$.

Proof. We prove this theorem again by induction on the structure of the input list.

Base case: $input = []$. 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, $11 \notin Output(N') = [0]$.

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 input after processing $input'$. By the induction hypothesis, we know $11\ \notin Output(N')$ and we must prove that $11\ \notin Output(N'')$.

Because we know that a neuron produces only 0 and 1 as output values, we know that $Output(N'') = Output(N') + [0]$ or $Output(N'') = Output(N') + [1]$. For the first case, we are done, because when *11* does not appear in a sequence, then by adding a 0 to the end of that sequence, there will still no 11 in that sequence.

The second case here is a bit more complicated. We need to split this case into two subcases. First, let's assume that the last produced output in $Output(N')$ is 0, i.e., $Output(N')$ has the form $Seq + [0]$. So, it is clear that by adding a 1 to a sequence which ended with 0 and didn't have any 11, the resulting sequence will not have any 11 as a substring. Thus, we can conclude that $11 \notin Output(N') \rightarrow 11 \notin Seq + [0] \rightarrow 11 \notin Seq + [0] + [1] \rightarrow 11 \notin Output(N') + [1] \rightarrow 11 \notin Output(N'')$.

Now for the second subcase, let's assume that the last produced output in $Output(N')$ is 1, i.e., $Output(N')$ has the form $Seq + [1]$. In this case, we have to prove that the next output will be 0. Because the last produced output in $Output(N')$ is 1, we know that $p(N') \geq \tau(N')$. So, $p(N'') = w_1(N').h$ and because $h = 0$ or $h = 1$, we can conclude that $p(N'') = w_1(N')$ or $p(N'') = 0$. In the first case, according to the property assumption, we know that $w_1(N) < \tau(N)$, and thus $p(N'') = w_1(N') = w_1(N) < \tau(N)$, and in the second case, because $\tau(N)$ is a positive value we have $p(N'') = 0 < \tau(N)$. Thus, by Equation (2), the next output produced will be 0. Therefore, $11 \notin Output(N') \rightarrow 11 \notin Seq + [1] \rightarrow 11 \notin Seq + [1] + [0] \rightarrow 11 \notin Output(N') + [1] \rightarrow 11 \notin Output(N'')$.

This completes the proof.

Property 3. $\forall\ (N{:}\,neuron)\ (input{:}\,list\ nat)$,

$length\big(w(N)\big) = 1 \wedge w_1(N) < 0 \rightarrow 1\ \notin Output(N')$

Similar to Property 2, in the statement above, $1 \notin Output(N')$ means there is no 1 in the list $Output(N')$.

Proof. We will prove this property using induction on the input length again.

Base case: $input = []$. 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, $1 \notin Output(N') = [0]$.

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 input after processing $input'$. By the induction hypothesis, we know $1\ \notin Output(N')$ and we must prove that $1\ \notin Output(N'')$.

Let t be the time at which we produced the most recent output. So, $p(N'')$ corresponds to $p(t)$ and $p(N')$ corresponds to $p(t-1)$. Again, note that $\tau(N) = \tau(N') = \tau(N'')$ and similar equalities hold for r and w_1, so we use them interchangeably. Using the induction hypothesis, we know that $1 \notin Output(N')$. So, the last produced output in $Output(N')$ is 0. Thus, $p(N') = p(t-1) < \tau(N')$. That makes $p(t) = p(N'') = w_1(N'').h + r(N'').p(N')$. We need to consider two cases, which are $h = 0$ and $h = 1$.

In the first case, $p(N'') = r(N'').p(N')$. We can conclude that $p(N'') = r(N'').p(N') < \tau(N')$ because $p(N') < \tau(N')$ by the property assumption and it is multiplied by $r(N')$, the leak factor, which is between 0 and 1. Recall that $\tau(N')$ is a positive value. Thus, the next output will be 0 in this case and $1 \notin Output(N') + [0] = Output(N'')$.

In the second case, $p(N'') = w_1(N'') + r(N'').p(N')$. With the same reasoning as the previous case, we can say that $r(N'').p(N') < \tau(N')$. Because $w_1(N')$ is a negative value, adding it to the left side of the inequality will make it smaller and the inequality still holds. Thus, $p(N'') = w_1(N'') + r(N'').p(N') < \tau(N')$. Therefore, the next output will be 0 in this case too and $1 \notin Output(N') + [0] = Output(N'')$.

This completes the proof.

Property 4. $\forall\ (Series{:}\,list\ Neuron)\ (input{:}\,list\ nat)\ (i{:}\,nat)$, $length(Series) = N \wedge length(w(Series[i])) = 1\ \wedge$ $w_1(Series[i]) > \tau(Series[i]) \rightarrow Output = zeros(N) + input$

Proof. This time we need to use induction on the length of $Series$. Here, we consider the series structure shown in Figure 1(a). In the formula above, $Series[i]$ denotes the i-th neuron in the $Series$ and

$zeros(N)$ means a sequence of 0s of length N. Also, $Output$ denotes the final output of the series structure which is equal to $Output(Series[N])$.

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 1 proved earlier, we can conclude that $Output = input + [0] = input + zeros(1)$.

Induction case: We assume that the property holds for $Series$ of length k. Let $Output'$ be the output of this series. Thus by the induction hypothesis, $Output' = zeros(k) + input$. We must show that the property holds for a $Series + [M]$, 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 $Series$, which is $zeros(k) + input$. By the assumptions of this property, all neurons in $Series + [M]$ satisfy Property 1, 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.