# Spiking Neural Networks modelled as Timed Automata with parameter learning

Elisabetta de Maria, Cinzia × Di Giusto, Laetitia Laversa

**Noname manuscript No.**
(will be inserted by the editor)

# Spiking Neural Networks modelled as Timed Automata

## with parameter learning

**Elisabetta De Maria · Cinzia Di Giusto · Laetitia Laversa**

Received: date / Accepted: date

**Abstract** In this paper we present a novel approach to automatically infer parameters of spiking neural networks. Neurons are modelled as timed automata waiting for inputs on a number of different channels (synap-ses), for a given amount of time (the accumulation period). When this period is over, the current *potential* value is computed considering current and past inputs. If this potential overcomes a given *threshold*, the automaton emits a broadcast signal over its output channel, otherwise it restarts another accumulation period. After each emission, the automaton remains inactive for a fixed *refractory period*. Spiking neural networks are formalised as sets of automata, one for each neuron, running in parallel and sharing channels according to the network structure. Such a model is formally validated against some crucial properties defined via proper temporal logic formulae. The model is then exploited to find an assignment for the synaptical weights of neural networks such that they can reproduce a given behaviour. The core of this approach consists in identifying some correcting actions adjusting synaptical weights and back-propagating them until the expected behaviour is displayed. A concrete case study is discussed.

**Keywords** Neural Networks · Parameter Learning · Timed Automata, Temporal Logic · Model Checking.

## 1 Introduction

The brain behaviour is the object of thorough studies: researchers are interested not only in the inner functioning of neurons (which are its elementary components), their interactions and the way these aspects participate to the ability to move, learn or remember, typical of living beings; but also in reproducing such capabilities (emulating nature), e.g., within robot controllers, speech/text/face recognition applications, etc. In order to achieve a detailed understanding of the brain functioning, both neurons behaviour and their interactions must be studied. Several models of the neuron behaviour have been proposed: some of them

Université Côte d'Azur, CNRS, I3S, France

make neurons behave as binary threshold gates, other ones exploit a sigmoidal transfer function, while, in many cases, differential equations are employed. According to [29,26], three different and progressive *generations* of neural networks can be recognised: (i) *first generation* models handle discrete inputs and outputs and their computational units are threshold-based transfer functions; they include McCulloch and Pitt's threshold gate model [28], the perceptron model[15], Hopfield networks [21], and Boltzmann machines [1]; (ii) *second generation* models exploit real valued activation functions, e.g., the sigmoid function, accepting and producing real values: a well known example is the multi-layer perceptron [8,32]; (iii) *third generation* networks are known as spiking neural networks. They extend second generation models treating time-dependent and real valued signals often composed by *spike trains*. Neurons may fire output spikes according to threshold-based rules which take into account input spike magnitudes and occurrence times [29].

The core of our analysis are *spiking neural networks* [16]. Because of the introduction of timing aspects they are considered closer to the actual brain functioning than other generations models. Spiking neurons emit spikes taking into account input impulses strength and their occurrence instants. Models of this sort are of great interest, not only because they are closer to natural neural networks behaviour, but also because the temporal dimension allows to represent information according to various *coding schemes* [30,29]: e.g., the amount of spikes occurred within a given time window (*rate* coding), the reception/absence of spikes over different synapses (*binary* coding), the relative order of spikes occurrences (*rate rank* coding), or the precise time difference between any two successive spikes (*timing* coding).

Several spiking neuron models have been proposed in the literature, having different complexities and capabilities. In [24], Izhikevich classifies spiking neuron models according to some *behaviour* (i.e., typical responses to an input pattern) that they should exhibit in order to be considered biologically relevant. The leaky integrate & fire (LI&F) model [25], where past inputs relevance exponentially decays with time, is one of the most studied neuron models because it is straightforward and easy to use [24,29]. On the other end of the spectrum, the Hodgkin-Huxley (H-H) model [20] is one of the most complex being composed by four differential equations comparing neurons to electrical circuits. In [24], the H-H model can reproduce all behaviours under consideration, but the simulation process is really expensive even for just a few neurons being simulated for a small amount of time. Our aim is to produce a neuron model being meaningful from a biological point of view but also amenable to formal analysis and verification, that could be therefore used to detect non-active portions within some network (i.e., the subset of neurons not contributing to the network outcome), to test whether a particular output sequence can be produced or not, to prove that a network may never be able to emit, to assess if a change to the network structure can alter its behaviour, or to investigate (new) learning algorithms which take time into account.

In this paper we focus on the *leaky integrate & fire* (LI&F) model originally proposed in [25]. It is a computationally efficient approximation of single-compartment model [24] and is abstracted enough to be able to apply formal verification techniques such as model-checking. Here we work on an extended version of the discretised formulation proposed in [13], which relies on the notion of logical time. Time

is considered as a sequence of logical 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. The variant we introduce here takes into account some new time-related aspects, such as a lapse of time in which the neuron is not active, i.e., it cannot receive and emit. We encode LI&F networks into timed automata: we show how to define the behaviour of a single neuron and how to build a network of neurons. Timed automata [2] are finite state automata extended with timed behaviours: constraints are allowed to limit the amount of time an automaton can remain within a particular state, or the time interval during which a particular transition may be enabled. Timed automata networks are sets of automata that can synchronise over *channel* communications.

Our modelling of spiking neural networks consists of timed automata networks where each neuron is an automaton. Its behaviour consists in accumulating the weighted sum of inputs, provided by a number of ingoing weighted synapses, for a given amount of time. Then, if the *potential* accumulated during the last and previous accumulation periods overcomes a given threshold, the neuron fires an output over the outgoing synapse. Synapses are channels shared between the timed automata representing neurons, while *spike* emissions are represented by *broadcast synchronisations* occurring over such channels. Timed automata are also exploited to produce or *recognise* precisely defined spike sequences.

As a first main contribution, we analyse some intrinsic properties of the proposed model, e.g., the maximum threshold value allowing a neuron to emit, or the lack of inter-spike memory, preventing the behaviour of a neuron from being influenced by what happened before the last spike. Furthermore, we encode in temporal logics all the behaviours (or capabilities) a LI&F model should be able to reproduce according to Izhikevich and we exploit model checking to prove these behaviours are reproducible in our model. Izhikevich also identifies a set of behaviours which are not expected to be reproducible by any LI&F model. We prove these limits to hold for our model, too, and we provide, for each non-reproducible behaviour, an extension of the model allowing to reproduce it.

As a second main contribution, we exploit our automata-based modelling to propose a new methodology for parameter inference in spiking neural networks. In particular, our approach allows to find an assignment for the synaptical weights of a given neural network such that it can reproduce a given behaviour. We apply the proposed approach to find suitable parameters in mutual inhibition networks, a well studied class of networks in which the constituent neurons inhibit each other neuron's activity [27].

This paper is an extended and revised version of the conference papers [10] and [11]. In particular, in section 5 we propose a refined version of the Advice Back-Propagation algorithm. Furthermore, we add a second learning technique that is based on simulation instead of model checking.

The rest of the paper is organised as follows: in Section 2 we recall definitions of timed automata networks, temporal logics, and model checking; in Section 3 we describe our reference model, the LI&F one, and its encoding into timed automata networks; in Section 4 we study some intrinsic properties of the obtained model and we validate it against its ability of reproducing or not some behaviours; in Section 5 we develop the novel parameter learning approach and we introduce a case study; in Section 6 we give an overview of the related work. Finally, Section 7 summarises our contribution and presents some future research directions.

## 2 Preliminaries

In this section we introduce the formal tools we adopt in the rest of the paper, namely timed automata and temporal logics.

### 2.1 Timed Automata.

Timed automata [2] are a powerful theoretical formalism for modelling and verifying real time systems. A timed automaton is an annotated directed (and connected) graph, with an initial node and provided with a finite set of non-negative real variables called *clocks*. Nodes (called *locations*) are annotated with *invariants* (predicates allowing to enter or stay in a location), arcs with *guards*, *communication labels*, and possibly with some variables upgrades and clock *resets*. Guards are conjunctions of elementary predicates of the form $x\ op\ c$, where $op \in \{>, \geq, =, <, \leq\}$, $x$ is a clock, and $c$ a (possibly parameterised) positive integer constant. As usual, the empty conjunction is interpreted as true. The set of all guards and invariant predicates will be denoted by $G$.

**Definition 1** A *timed automaton* $TA$ is a tuple $(L, l^0, X, \Sigma, Arcs, Inv)$, where

- $L$ is a set of locations with $l^0 \in L$ the initial one
- $X$ is the set of clocks,
- $\Sigma$ is a set of communication labels,
- $Arcs \subseteq L \times (G \cup \Sigma \cup U) \times L$ is a set of arcs between locations with a guard in $G$, a communication label in $\Sigma \cup \{\varepsilon\}$, and a set of variable upgrades (e.g., clock resets);
- $Inv : L \to G$ assigns invariants to locations.

It is possible to define a synchronised product of a set of timed automata that work and synchronise in parallel. The automata are required to have disjoint sets of locations, but may share clocks and communication labels which are used for synchronisation. We restrict communications to be *broadcast* through labels $b!, b? \in \Sigma$, meaning that a set of automata can synchronise if one is emitting; notice that a process can always emit (e.g., $b!$) and the receivers ($b?$) must synchronise if they can.

Locations can be normal, urgent or committed. Urgent locations force the time to freeze, committed ones freeze time and the automaton must leave the location as soon as possible, i.e., they have higher priority.

The synchronous product $TA_1 \parallel \ldots \parallel TA_n$ of timed automata, where $TA_j = (L_j, l_j^0, X_j, \Sigma_j, Arcs_j, Inv_j)$ and $L_j$ are pairwise disjoint sets of locations for each $j \in [1, \ldots, n]$, is the timed automaton

$$TA = (L, l^0, X, \Sigma, Arcs, Inv)$$

such that:

- $L = L_1 \times \ldots \times L_n$ and $l^0 = (l_1^0, \ldots, l_n^0)$, $X = \bigcup_{j=1}^{n} X_j$, $\Sigma = \bigcup_{j=1}^{n} \Sigma_j$,
- $\forall l = (l_1, \ldots, l_n) \in L$: $Inv(l) = \bigwedge_j Inv_j(l_j)$,

– *Arcs* is the set of arcs $(l_1, \ldots, l_n) \xrightarrow{g,a,r} (l'_1, \ldots, l'_n)$ such that for all $1 \leq j \leq n$ then $l'_j = l_j$.

Its semantics is the one of the underlying timed automaton $TA$ with the following notations. A location is a vector $l = (l_1, \ldots, l_n)$. We write $l[l'_j/l_j, j \in S]$ to denote the location $l$ in which the $j^{th}$ element $l_j$ is replaced by $l'_j$, for all $j$ in some set $S$. A valuation is a function $\nu$ from the set of clocks to the non-negative reals. Let $\mathbb{V}$ be the set of all clock valuations, and $\nu_0(x) = 0$ for all $x \in X$. We shall denote by $\nu \vDash F$ the fact that the valuation $\nu$ satisfies (makes true) the formula $F$. If $r$ is a clock reset, we shall denote by $\nu[r]$ the valuation obtained after applying the clock reset $r \subseteq X$ to $\nu$; and if $d \in \mathbb{R}_{>0}$ is a delay, $\nu + d$ is the valuation such that, for any clock $x \in X$, $(\nu + d)(x) = \nu(x) + d$.

The semantics of a synchronous product $TA_1 \parallel \ldots \parallel TA_n$ is defined as a timed transition system $(S, s_0, \rightarrow)$, where $S = (L_1 \times, \ldots \times L_n) \times \mathbb{V}$ is the set of states, $s_0 = (l^0, \nu_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

– (silent): $(l, \nu) \rightarrow (l', \nu')$ if there exists $l_i \xrightarrow{g,\varepsilon,r} l'_i$, for some $i$, such that $l' = l[l'_i/l_i]$, $\nu \vDash g$ and $\nu' = \nu[r]$,
– (broadcast): $(\bar{l}, \nu) \rightarrow (\bar{l'}, \nu')$ if there exists an output arc $l_j \xrightarrow{g_j,b!,r_j} l'_j \in Arcs_j$ and a (possibly empty) set of input arcs of the form $l_k \xrightarrow{g_k,b?,r_k} l'_k \in Arcs_k$ such that for all $k \in K = \{k_1, \ldots, k_m\} \subseteq \{l_1, \ldots, l_n\} \backslash \{l_j\}$, the size of $K$ is maximal, $\nu \vDash \bigwedge_{k \in K \cup \{j\}} g_k$, $l' = l[l'_k/l_k, k \in K \cup \{j\}]$ and $\nu' = \nu[r_k, k \in K \cup \{j\}]$;
– (timed): $(l, \nu) \rightarrow (l, \nu + d)$ if $\nu + d \vDash Inv(l)$.

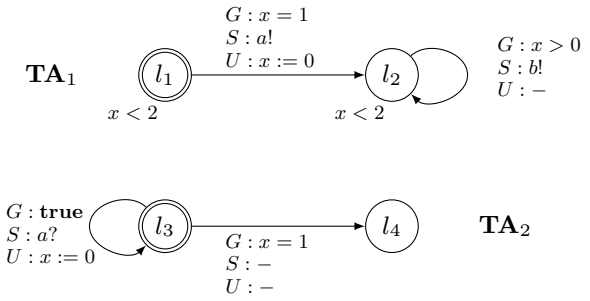
The valuation function $\nu$ is extended to handle a set of shared bounded integer variables: predicates concerning such variables can be part of edges guards or locations invariants, moreover variables can be updated on edges firings but they cannot be assigned to or from clocks.

*Example 1* In Figure 1 we consider the network of timed automata $TA_1$ and $TA_2$ with broadcast communications, and we give a possible run. $TA_1$ and $TA_2$ start in the $l_1$ and $l_3$ locations, respectively, so the initial state is $[(l_1, l_3);\ x = 0]$. A *timed* transition produces a delay of 1 time unit, making the system move to state $[(l_1, l_3);\ x = 1]$. A *broadcast* transition is now enabled, making the system move to state $[(l_2, l_3);\ x = 0]$, broadcasting over channel $a$ and resetting the $x$ clock. Two successive *timed* transitions (0.5 time units) followed by a *broadcast* one will eventually lead the system to state $[(l_2, l_4);\ x = 1]$. ◇

Throughout our modelling, we have used the specification and analysis tool `Uppaal` [4], which provides the possibility of designing and simulating timed automata networks on top of the ability of testing networks against temporal logic formulae. All figures depicting timed automata follow the graphic conventions of the tool (e.g., initial states are denoted with a double circle).

### 2.2 Temporal Logics and Model Checking

Model checking is one of the most common approaches to the verification of software and hardware (distributed) systems [7]. It allows to automatically prove




(a) The timed automata network $TA_1 \parallel TA_2$.

$[(l_1, l_3);\ x = 0]$
↓
$[(l_1, l_3);\ x = 1]$
↓
$[(l_2, l_3);\ x = 0]$
↓
$[(l_2, l_3);\ x = 0.5]$
↓
$[(l_2, l_3);\ x = 1]$
↓
$[(l_2, l_4);\ x = 1]$

(b) A possible run.

Fig. 1: A network of timed automata with a possible run.

whether a system verifies or not a given specification. In order to apply such a technique, the system at issue should be encoded as a finite transition system and the specification should be written using propositional temporal logic. Formally, a transition system over a set $AP$ of atomic propositions is a tuple $M = (Q, T, L)$, where $Q$ is a finite set of states, $T \subseteq Q \times Q$ is a total transition relation, and $L : Q \to 2^{AP}$ is a labelling function that maps every state into the set of atomic propositions that hold at that state.

Temporal formulae describe the dynamical evolution of a given system. The computation tree logic CTL$^*$ allows to describe properties of computation trees. Its formulas are obtained by (repeatedly) applying boolean connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$), *path quantifiers*, and *state quantifiers* to atomic formulas. The path quantifier **A** (resp., **E**) can be used to state that all the paths (resp., some path) starting from a given state have some property. The state quantifiers are **X** (next time), which specifies that a property holds at the next state of a path, **F** (sometimes in the future), which requires a property to hold at some state on the path, **G** (always in the future), which imposes that a property is true at every state on the path, and **U** (until), which holds if there is a state on the path where the second of its argument properties holds and, at every preceding state on the path, the first of its two argument properties holds. Given two formulas $\varphi_1$ and $\varphi_2$, in the rest of the paper we use the shortcut $\varphi_1 \rightsquigarrow \varphi_2$ to denote the liveness property $AG(\varphi_1 \rightarrow AF\varphi_2)$, which can be read as "$\varphi_1$ always leads to $\varphi_2$ ".

The branching time logic CTL is a fragment of CTL$^*$ that allows quantification over the paths starting from a given state. Unlike CTL$^*$, it constrains every state quantifier to be immediately preceded by a path quantifier.

Given a transition system $M = (Q, T, L)$, a state $q \in Q$, and a temporal logic formula $\varphi$ expressing some desirable property of the system, the *model checking problem* consists of establishing whether $\varphi$ holds at $q$ or not, namely, whether $M, q \models \varphi$.

## 3 Leaky Integrate and Fire Model and Mapping to Timed Automata

Spiking neural networks [26] are modelled as directed weighted graphs where vertices are computational units and edges represent *synapses*. The signals propagating over synapses are trains of impulses: *spikes*. Synapses may modulate these signals according to their weight: *excitatory* if positive, or *inhibitory* if negative.

The dynamics of neurons is governed by their *membrane potential* (or, simply, *potential*), representing the difference of electrical potential across the cell membrane. The membrane potential of each neuron depends on the spikes received over the ingoing synapses. Both current and past spikes are taken into account, even if old spikes contribution is lower. In particular, the *leak factor* is a measure of the neuron memory about past spikes. The neuron outcome is controlled by the algebraic difference between its membrane potential and its *firing threshold*: it is enabled to fire (i.e., emit an output impulse over *all* outgoing synapses) only if such a difference is non-negative. Spike propagation is assumed to be instantaneous. Immediately after each emission the neuron membrane potential is reset and the neuron stays in a *refractory period* for a given amount of time. During this period it has no dynamics: it cannot increase its potential as any received spike is lost and therefore it cannot emit any spike.

**Definition 2 (Spiking Integrate and Fire Neural Network)** *A spiking integrate and fire neural network* is a tuple $(V, A, w)$, where:

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

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 spiking integrate and fire neuron* $v$ is characterized by a parameter tuple

$$(\theta_v, \tau_v, \lambda_v, p_v, y_v),$$

where:

- $\theta_v \in \mathbb{N}$ is the *firing threshold*,
- $\tau_v \in \mathbb{N}^+$ is the *refractory* period,
- $\lambda_v \in \mathbb{Q} \cap [0, 1]$ is the *leak factor*.

The dynamics of a *spiking integrate and fire neuron* $v$ is given by:

- $p_v : \mathbb{N} \to \mathbb{Q}_0^+$ is the [membrane] *potential* function defined as

$$p_v(t) = \begin{cases} \sum_{i=1}^{m} w_i \cdot x_i(t), & \text{if } p_v(t-1) \geqslant \theta_v \\ \sum_{i=1}^{m} w_i \cdot x_i(t) + \lambda_v \cdot p_v(t-1), & \text{otherwise.} \end{cases}$$

  with $p_v(0) = 0$ 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),

– $y_v : \mathbb{N} \to \{0, 1\}$ is the neuron output function, defined as

$$y_v(t) = \begin{cases} 1 & \text{if } p_v(t) \geqslant \theta_v \\ 0 & \text{otherwise.} \end{cases}$$

As shown in the previous definition, the set of neurons of a spiking integrate and fire neural network can be classified into input, intermediary, and output ones. Each input neuron can only receive as input external signals (and not other neurons' output). 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.

We present here our modelling of spiking integrate and fire neural networks (in the following denoted as neural networks) via timed automata networks. Let $S = (V, A, w)$ be a neural network, $G$ be a set of *input generator* neurons (these fictitious neurons are connected to input neurons and generate input sequences for the network), and $O$ be a set of *output consumer* neurons (these fictitious neurons are connected to the broadcast channel of each output neuron and aim at consuming their emitted spikes). The corresponding timed automata network is obtained as the synchronous product of the encoding of input generator neurons, the neurons of the network (referred as standard neurons in the following), and output consumers neurons. More formally:

$$[\![S]\!] = \left( \Big\|_{n_g \in G} [\![n_g]\!] \right) \| \left( \Big\|_{v_j \in V} [\![v_j]\!] \right) \| \left( \Big\|_{n_c \in O} [\![n_c]\!] \right)$$

**Input generators.** The behaviour of input generator neurons is part of the specification of the network. Here we define two kinds of input behaviours: regular and non-deterministic ones. For each family, we provide an encoding into timed automata.

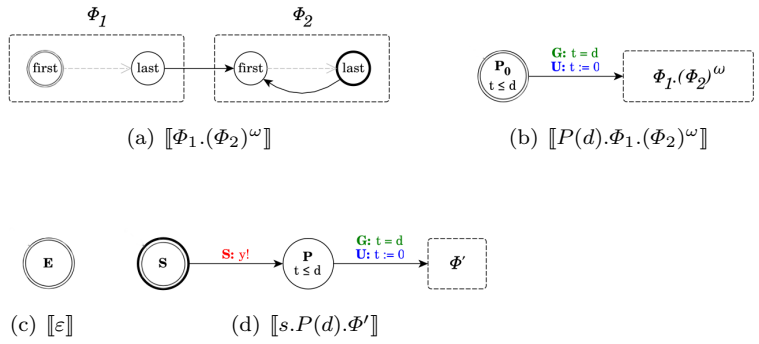
*Regular input sequences.* Spike trains are "regular" sequences of spikes and pauses: spikes are instantaneous while pauses have a non-null duration. Sequences can be *empty*, *finite* or *infinite*. After each spike there must be a pause, except when the spike is the last event of a finite sequence. Infinite sequences are composed by two parts: a finite and arbitrary prefix and an infinite and periodic part composed by a finite sequence of *spike–pause* pairs which is repeated infinitely often. More formally, such sequences are given in terms of the following grammar:

$$\begin{aligned} G &::= \Phi.(\Phi)^\omega \mid P(d).\Phi.(\Phi)^\omega \\ \Phi &::= s.P(d).\Phi \mid \varepsilon \end{aligned}$$

with $s$ representing a spike and $P(d)$ a pause of duration $d$. It is possible to generate an emitter automaton for any regular input sequence:

**Definition 3 (Input generator)** Let $I \in \mathcal{L}(G)$ be a word over the language generated by $IS$, then its encoding into timed automata is $[\![I]\!] = (L(I), first(I), \{t\}, \{y\}, Arcs(I), Inv(I))$. It is inductively defined as follows:

– if $I := \Phi_1.(\Phi_2)^\omega$
  – $L(I) = L(\Phi_1) \cup L(\Phi_2)$, where $last(\Phi_2)$ is *urgent*
  – $first(I) = first(\Phi_1)$




(a) $[\![\Phi_1.(\Phi_2)^\omega]\!]$ (b) $[\![P(d).\Phi_1.(\Phi_2)^\omega]\!]$

(c) $[\![\varepsilon]\!]$ (d) $[\![s.P(d).\Phi']\!]$

Fig. 2: Representation of the encoding of an input sequence

  - $Arcs(I) = Arcs(\Phi_1) \cup Arcs(\Phi_2) \cup$
    $\{(last(\Phi_1), \mathbf{true}, \varepsilon, \emptyset, first(\Phi_2)),$
    $(last(\Phi_1), \mathbf{true}, \varepsilon, \emptyset, first(\Phi_2))\}$
  - $Inv(I) = Inv(\Phi_1) \cup Inv(\Phi_2)$
- if $I := P(d).\Phi_1.(\Phi_2)^\omega$
  - $L(I) = \{\mathbf{P}_0\} \cup L(\Phi_1) \cup L(\Phi_2)$, where $last(\Phi_2)$ is *urgent*
  - $first(I) = \mathbf{P}_0$
  - $Arcs(I) = Arcs(\Phi_1) \cup Arcs(\Phi_2) \cup$
    $\{(\mathbf{P}_0, t \leq d, \ , \{t := 0\}, first(\Phi_1)),$
    $(last(\Phi_1), \mathbf{true}, \varepsilon, \emptyset, first(\Phi_2)),$
    $(last(\Phi_1), \mathbf{true}, \varepsilon, \emptyset, first(\Phi_2))\}$
  - $Inv(I) = \{\mathbf{P}_0 \mapsto t \leq d\} \cup Inv(\Phi_1) \cup Inv(\Phi_2)$
- if $\Phi := \varepsilon$
  - $L(\Phi) = \{\mathbf{E}\}$
  - $first(\Phi) = last(\Phi) = \mathbf{E}$
  - $Arcs(\Phi) = \emptyset$
  - $Inv(\Phi) = \emptyset$
- if $\Phi := s.P(d).\Phi'$
  - $L(\Phi) = \{\mathbf{S}, \mathbf{P}\} \cup L(\Phi')$
  - $first(\Phi) = \mathbf{S}, \quad last(\Phi) = last(\Phi')$
  - $Arcs(\Phi) = Arcs(\Phi') \cup \{(\mathbf{S}, \mathbf{true}, y!, \emptyset, \mathbf{P}),$
    $(\mathbf{P}, t = d, \varepsilon, \{t := 0\}, first(\Phi'))\}$
  - $Inv(\Phi) = \{\mathbf{P} \mapsto t \leq d\} \cup Inv(\Phi')$

Figure 2 depicts the shape of input generators. Figure 2(a) shows the generator $[\![I]\!]$, obtained from $I := \Phi_1.(\Phi_2)^\omega$. The edge connecting the last state of $[\![\Phi_2]\!]$ to the first one allows $\Phi_2$ to be repeated infinitely often. Figure 2(b) shows the case of an input sequence $I := P(d).\Phi_1.(\Phi_2)^\omega$ beginning with a pause $P(d)$: in this case, the initial location of $[\![I]\!]$ is $\mathbf{P}_0$, which imposes a delay of $d$ time units. The remainder of the input sequence is encoded as for the previous case. Figure 2(c) shows the induction basis for encoding a sequence $\Phi$, i.e., the case $\Phi := \varepsilon$. It is encoded as a location $\mathbf{E}$ having no edge. Finally, Figure 2(d) shows the case of a non-empty spike–pause pair sequence $\Phi := s.P(d).\Phi'$. It consists of an *urgent* location $\mathbf{S}$: when the automaton moves from $\mathbf{S}$, a spike is fired over channel $y$ and

the automaton moves to location **P**, representing a silent period. After that, the automaton proceeds with the encoding of $\Phi'$.

*Non-deterministic input sequences.* This kind of input sequences is useful when no assumption is available on neuron inputs. These are random sequences of spikes separated by at least $T_{min}$ time units.

Such sequences can be generated by an automaton defined as follows:

**Definition 4 (Non-deterministic input generator)** A non-deterministic input generator $I_{nd}$ is a tuple

$$(L, \mathbf{B}, X, \Sigma, Arcs, Inv),$$

with:

- $L = \{\mathbf{B}, \mathbf{S}, \mathbf{W}\}$, with **S** urgent,
- $X = \{t\}$
- $\Sigma = x$
- $Arcs = \{(\mathbf{B}, t = D, x!, \emptyset, \mathbf{S}), (\mathbf{S}, \mathbf{true}, \varepsilon, \{t := 0\}, \mathbf{W}), (\mathbf{W}, t > T_{min}, x!, \emptyset, \mathbf{S})\}$
- $Inv(B) = (t \leq D)$

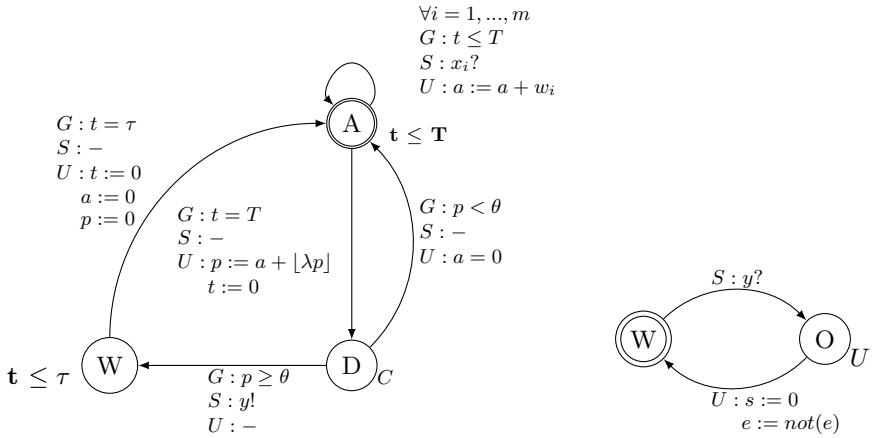
where $D$ is the initial delay.

The behavior of such a generator depends on clock $t$ and broadcast channel $x$, and can be summarized as follows: it waits in location **B** an arbitrary amount of time before moving to location **S**, firing its first spike over channel $x$. Since location **S** is *urgent*, the automaton instantaneously moves to location **W**, resetting clock $t$. Finally, from location **W**, after an arbitrary amount of time $t$, it moves to location **S**, firing a spike. Notice that an initial delay $D$ may be introduced by adding the invariant $t \leq D$ to the location **B** and the guard $t = D$ on the edge $(\mathbf{B} \rightarrow \mathbf{S})$.

**Standard neurons.** The neuron is a computational unit behaving as follows: i) it accumulates potential whenever it receives input spikes within a given *accumulation period*, ii) if the accumulated potential is greater than the *threshold*, it emits an output spike, iii) it waits during a *refractory period*, and restarts from i). Observe that the accumulation period is not present in the definition of neuron (Definition 2). It is indeed introduced here to slice time and therefore discretise the decrease of the potential value due to the leak factor. We assume that two input spikes on the same synapse cannot be received within the same accumulation period (i.e., the accumulation period is shorter than the minimum refractory period of the input neurons of the network). Next, we give the encoding of neurons into timed automata.

**Definition 5** Given a neuron $v = (\theta, \tau, \lambda, p, y)$ with $m$ input synapses, its encoding into timed automata is $\mathcal{N} = (L, \mathbf{A}, X, Var, \Sigma, Arcs, Inv)$ with:

- $L = \{\mathbf{A}, \mathbf{W}, \mathbf{D}\}$ with **D** committed,
- $X = \{t\}$
- $Var = \{p, a\}$
- $\Sigma = \{x_i \mid i \in [1..m]\} \cup \{y\}$,




(a) Neuron model.

(b) Output consumer automaton.

Fig. 3: Automata for standard neuron and output consumer.

- $Arcs = \{(\mathbf{A}, t \leq T, x_i?, \{a := a + w_i\}, \mathbf{A}) \mid i \in [1..m]\} \cup \{(\mathbf{A}, t = T, \ , \{p := a + \lfloor \lambda p \rfloor\}, \mathbf{D}),$
  $(\mathbf{D}, p < \theta, \ , \{a := 0\}, \mathbf{A}), (\mathbf{D}, p \geq \theta, y!, \ , \mathbf{W}),$
  $(\mathbf{W}, t = \tau, \ , \{a := 0, t := 0, p := 0\}, \mathbf{A})\}$ ;
- $Inv(\mathbf{A}) = t \leq T, Inv(\mathbf{W}) = t \leq \tau, Inv(\mathbf{D}) = \mathbf{true}$.

The neuron behavior, described by the automaton in Figure 3(a), depends on the following channels, variables and clocks:

- $x_i$ for $i \in [1..m]$ are the $m$ input channels,
- $y$ is the broadcast channel used to emit the output spike,
- $p \in \mathbb{N}$ is the current potential value, initially set to zero,
- $a \in \mathbb{N}$ is the weighted sum of input spikes occurred within the *current* accumulation period; it equals zero at the beginning of each round.

The behaviour of the automaton modelling neuron $v$ can be summed up as follows:

- the neuron keeps waiting in state **A** (for Accumulation) for input spikes while $t \leqslant T$ and, whenever it receives a spike on input $x_i$, it updates $a$ with $a := a + w_i$;
- when $t = T$, the neuron moves to state **D** (for Decision), resetting $t$ and updating $p$ according to the potential function given in Definition 2:

$$p := a + \lfloor \lambda \cdot p \rfloor$$

  Since state **D** is *committed*, it does not allow time to progress, so, from this state, the neuron can move back to state **A** resetting $a$ if the potential has not reached the threshold $p < \theta$, or it can move to state **W**, firing an output spike, otherwise;
- the neuron remains in state **W** (for Wait) for $\tau$ time units ($\tau$ is the length of the refractory period) and then it moves back to state **A** resetting $a$, $p$ and $t$.

**Output consumers.** In order to have a complete modelling of a spiking neural network, for each output neuron we build an *output consumer* automaton $O_y$. The automaton, whose formal definition is straightforward, is shown in Figure 3(b). The consumer waits in location **W** for the corresponding output spikes on channel $y$ and, as soon as it receives the spike, it moves to location **O**. This location is only needed to simplify model checking queries. Since it is urgent, the consumer instantly moves back to location **W** resetting $s$, the clock measuring the elapsed time since last emission, and setting $e$ to its negation, with $e$ being a *boolean variable* which differentiates each emission from its successor.

**Definition 6 (Output consumer)** An output consumer is a timed automaton

$$\mathcal{N} = (L, \mathbf{W}, X, Var, \Sigma, Arcs, Inv)$$

with:

- $L = \{\mathbf{W}, \mathbf{O}\}$ with O urgent,
- $X = \{s\}$
- $Var = \{e\}$
- $\Sigma = \{y_i \mid y_i \text{ is an output neuron}\}$
- $Arcs = \{(\mathbf{W}, , y?, , \mathbf{O}),$
  $(\mathbf{O}, s := 0, , \{e := not(e)\}, \mathbf{W})\}$
- $Inv(\mathbf{W}) = \mathbf{true}, Inv(\mathbf{O}) = \mathbf{true}$.

We have a complete implementation of the spiking neural network model proposed in the paper via the tool `Uppaal`. It can be found on the web page [6]. We have validated our neuron model against some characteristic properties studied in [24] (tonic spiking, excitability, integrator, etc.). These properties have been formalised in temporal logics and checked via model-checking tools.

Observe that, since we rely on a discrete time, we could have used tick automata [17], a variant of Büchi automata where a special clock models the discrete flow of time. However, to the best of our knowledge, no existing tool allows to implement such automata. We decided to opt for timed automata in order to have an effective implementation of our networks to be exploited in parameter learning algorithms.

## 4 Validation of the model

In this section we show some properties of the neuron model of Definition 5. The first group of properties are structural. We can compute a minimum value such that any neuron, having a threshold greater than or equal to it, will never be able to fire.

*Property 1* Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron and $a_{max}$ be the maximum value received during each accumulation period. Then, if $\theta \geq \frac{a_{max}}{1-\lambda}$, the neuron is not able to fire.

*Proof* Without loss of generality, we suppose that, during each accumulation period, $\mathcal{N}$ receives the maximum possible input $a_{max}$. Then, its potential function is:

$$p_n = a_{max} + \lfloor \lambda \cdot p_{n-1} \rfloor$$

which is always lower than or equal to its undiscretized version:

$$p_n \leq {p'}_n = a_{max} + \lambda \cdot p'_{n-1}$$

The same inequality can be written in explicit form:

$$p_n \leq {p'}_n = \sum_{k=0}^{n} a_{n-k} \cdot \lambda^k$$

and, since we assumed the neuron always receives $a_{max}$, $a_{n-k}$ is constant and does not depend on $k$:

$$p_n \leq a_{max} \cdot \sum_{k=0}^{n} \lambda^k$$

The rightmost factor is a geometric series:

$$p_n \leq a_{max} \cdot \frac{1-\lambda^n}{1-\lambda}$$

which reaches its maximum value $\frac{1}{1-\lambda}$ for $n \to \infty$, therefore:

$$p_n \leq \frac{a_{max}}{1-\lambda}.$$

Thus, if $\theta \geq \frac{a_{max}}{1-\lambda}$, it is impossible for the neuron potential to reach the threshold and, consequently, the neuron cannot fire. □

In what follows, we only consider neurons that respect the previous constraint.

Apart from the minimum threshold, we can also quantify the amount of time that the neuron requires to complete an accumulate–fire–rest cycle. We show that there exists a minimum delay between neuron emissions.

*Property 2* Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron. Then the time difference between successive firings cannot be lower than $T + \tau$.

*Proof* Let $A_n = \sum_{k=1}^{T} a_{k+t_0}$ be the sum of weighted inputs during the $n$-th *accumulation period*, then the neuron behaviour can be described as follows:

$$p_n = A_n + \lfloor \lambda \cdot p_{n-1} \rfloor$$

which is the potential value after the $n$-th accumulation period. If the neuron eventually fires an output spike, then there exists $\hat{n} > 0$ such that:

$$\hat{n} = \underset{n \in \mathbb{N}}{arg\,min} \{p_n : p_n \geq \theta\}$$

i.e., the firing will occur at the end of the $\hat{n}$-th accumulation period, which means during the $\hat{t}$-th time unit since $t_0$, thus:

$$\hat{t} = \hat{n} \cdot T + t_0$$

where $t_0$ is the *last* reset time, i.e., the last instant back in time when the neuron completed its refractory period. Then the *next* reset time $t'$, i.e., the next instant

in future when the neuron will complete its refractory period, after having emitted a spike, is:

$$t' = \hat{t} + \tau = \hat{n} \cdot T + \tau + t_0$$

At instant $t'$, the neuron quits its refractory period, $n$ is reset to 0, $t_0$ is set to $t'$, and $\hat{n}$, $\hat{t}$ and $t'$ must be consequently re-computed as described above.

Such a way to describe our model dynamics allow us to express the *inter-firing period* as a function of $\hat{n}$:

$$t' - t_0 = \hat{n} \cdot T + \tau$$

So, the minimum inter-firing period is $T + \tau$ for $\hat{n} = 1$. □

Such a property can also be verified as follows: let $\mathcal{I}$ be the non-deterministic input generator having $T_{min} = 1$ and, without loss of generality[1], let the initial delay $D = T + \tau$. Then the timed automata network $\mathcal{I} \| \mathcal{N} \| \mathcal{O}$ satisfies the following formula:

$$AG(state_{\mathcal{O}}(\mathbf{O}) \rightarrow eval_{\mathcal{O}}(s) \geq T + \tau)$$

where $s$ measures the time elapsed since last firing, meaning that, whenever the output consumer receives a spike, the time elapsed since the previous received spike cannot be lower than $T + \tau$.

The next fact states that only positive stimulations are necessary for the neuron to produce emissions.

**Fact 1** *Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron, $a(t)$ the sum of weighted inputs received during the current accumulation period, and $p(t-1)$ the neuron potential at the end of the previous accumulation period. If $p(t-1) < \theta$ and $a(t) < 0$, the neuron cannot fire at the end of the current accumulation period. Moreover, if $p(t) \geq \theta$ then $a(t) > 0$.*

The neuron potential is affected by every input spike it received *since the last reset time*, but every event that occurred before that instant is forgotten: i.e., neurons are memoryless.

**Definition 7 (Inter-emission memory)** Let $\mathcal{N}$ be a neuron, $Z_{\mathcal{N}}$ its reset times set, and $I$ an input sequence. Then $\mathcal{N}$ has inter-emission *memory* if and only if there exist two different $t, t' \in Z_{\mathcal{N}}$ such that the output sequences produced by $\mathcal{N}$ as a response to $I$ starting from $t$ and $t'$ are different.

*Property 3* Neurons have not inter-emission memory.

*Proof* When the neuron moves from location **W** to **A**, it resets clock $t$ and variables $p$ and $a$, making them equal to their initial values. This entails that the neuron, if subjected to the same input sequence, will always behave in the same way. □

Next, we validate the neuron model against its ability of reproducing or not some behaviours, as described by Izhikevich in [24]. We introduce first three behaviours that are verified by our model.

[1] the initial delay is required in order to make the formula hold for the first output spike too

**Tonic Spiking.** *Tonic spiking* is the behaviour of a neuron producing a periodic output sequence as a response to a persistent excitatory constant input sequence.

*Property 4 (Tonic spiking)* Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron having only one ingoing excitatory synapse of weight $w$ and let $\mathcal{I}$ be the input source connected to $\mathcal{N}$ producing a persistent input sequence. Then $\mathcal{N}$ produces a periodic output sequence.

The property holds by construction. It can be tested via model checking in the following way. Let $\mathcal{I}$ be the fixed-rate input generator having arbitrary initial delay $D$, and let $\mathcal{O}$ be an output consumer. Then the timed automata network $\mathcal{I}\|\mathcal{N}\|\mathcal{O}$ satisfies the following formulae:

$$\begin{cases} state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{O}}(e) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O}) \wedge \neg eval_{\mathcal{O}}(e) \\ state_{\mathcal{O}}(\mathbf{O}) \wedge \neg eval_{\mathcal{O}}(e) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{O}}(e) \end{cases}$$

where $\mathbf{O}$ is the location that the consumer automaton $\mathcal{O}$ reaches after consuming a spike and $e$ is an alternating boolean variable whose value flips whenever $\mathcal{O}$ moves into location $\mathbf{O}$. So, whenever automaton $\mathcal{O}$ reaches location $\mathbf{O}$, it will eventually reach it again.

One may also find the value $P$ of the period of some given neuron $\mathcal{N}$ by means of simulations, thus the periodic behaviour can be proven verifying the following formula:

$$AG(state_{\mathcal{O}}(\mathbf{O}) \wedge eval_{\mathcal{N}}(f) \rightarrow eval_{\mathcal{O}}(s) = P)$$

where $s$ is the clock measuring the time elapsed since last spike consumed by $\mathcal{O}$, and $f$ is a boolean variable of automaton $\mathcal{N}$ which is initially $false$ and is set to $true$ when edge $(\mathbf{W} \rightarrow \mathbf{A})$ fires (i.e., it indicates whether $\mathcal{N}$ has already emitted the first spike and waited the first refractory period or not).

**Integrator.** *Integrator* is the behaviour of a neuron producing an output spike whenever it receives *at least* a specific number of spikes from its input sources in the same accumulation period.

*Property 5 (Integrator)* Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron having $m$ synapses with maximum excitatory weight $R$ and a threshold $n \leq m$. Then the neuron emits if it receives a spike from at least $n$ input sources during the same accumulation period.

As in the previous case, we can use model checking tools and test the formula stating that, if at least $n$ generators are ready to emit (location $\mathbf{S}$) while $\mathcal{N}$ is in $\mathbf{A}$, then $\mathcal{O}$ will eventually capture an output of $\mathcal{N}$:

$$\left( \sum_{i=1}^{m} state_i(\mathbf{S}) \geq n \right) \wedge state_{\mathcal{N}}(\mathbf{A}) \rightsquigarrow state_{\mathcal{O}}(\mathbf{O})$$

Notice that, since potential depends on past inputs too, the neuron may still be able to fire in other circumstances, e.g., if it keeps receiving less than $n$ spikes for a sufficient number of accumulation periods, then it may eventually fire.

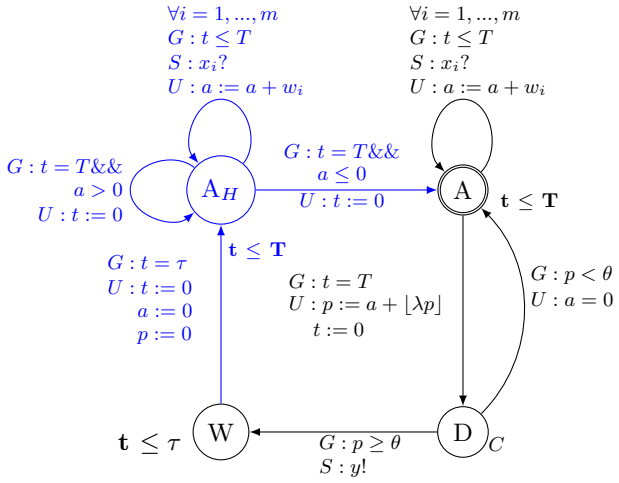



Fig. 4: The extended neuron model for phasic spiking.
Additions are colored in blue.

**Excitability.** *Excitability* is the behaviour of a neuron emitting sequences having a *decreasing* inter-firing period, i.e., an increasing output frequency, when stimulated by an *increasing* number of excitatory inputs.

*Property 6 (Excitability)* Let $\mathcal{N} = (\theta, \tau, \lambda, p, y)$ be a neuron having $m$ excitatory synapses. Then the inter-spike period decreases as the sum of weighted input spikes increases.

*Proof* If we assume the neuron is receiving an increasing number of excitatory spikes, generated by an increasing number of input sources emitting persistent inputs, then $a_t$ is the non-negative, non-decreasing and progressing (i.e., $\forall u\ \exists t : a_t > u$) succession representing the weighted sum of inputs within the $t$-th time unit. Consequently, $A_n = \sum_{k=1}^{T} a_{k+t_0}$ is the non-negative, non-decreasing and progressing succession counting the total sum of inputs within the $n$-th accumulation period. Since $A_n$ is positive and Property 2 holds, we can prove that the inter-spike period $t_n - t_{n-1}$ decreases. □

The following behaviours are not satisfied by the LI&F model, we show that our encoding cannot verify them as well.

**Phasic Spiking.** *Phasic spiking* is the behaviour of a neuron producing a *single* output spike when receiving a persistent and excitatory input sequence and then remaining quiescent for the rest of it. Such a behaviour depends on the neuron to have inter-emission memory.

*Property 7* Neurons cannot reproduce the *phasic spiking* behaviour.

*Proof* The phasic spiking behaviour requires the neuron to ignore any excitatory input spike occurring after its first emission. This means producing different outcomes, before and after the first emission, as a response to the same input sequence, which is impossible for a memoryless neuron, as stated in Property 3. □

We can extend our model to reproduce phasic spiking, see Figure 4. This variant makes the neuron able to "remember" if it is receiving a persistent excitatory input sequence. After each refractory period, the neuron moves to location $\mathbf{A_H}$, instead of $\mathbf{A}$. The only difference between $\mathbf{A_H}$ and $\mathbf{A}$ is that $\mathbf{A_H}$ *ignores* positive values of $a$ at the end of each accumulation period. Conversely, a non-positive value of $a$ (denoting the end of the persistent input), at the end of some accumulation period, leads the neuron back in location $\mathbf{A}$.

**Bursting.** A *burst* is a finite sequence of *high frequency* spikes. More formally:

**Definition 8** A spike output sequence is a *burst* if it is composed by spikes having an occurrence rate greater than $1/\tau$, with $\tau$ being the refractory period of the neuron.

A *burst sequence* is a sequence composed by bursts, subject to the following constraint: the time difference between the last spike of each burst and the first spike of the next burst it greater than $\tau$.

*Property 8* Neurons cannot produce bursts.

*Proof* A neuron $\mathcal{N}$ cannot emit spikes having a rate greater than $1/(T+\tau)$, as stated by Property 2, so it cannot produce bursts. □

In order to reproduce bursts our model can be extended by allowing several subsequent emissions in an interval period smaller than $\tau$. After this period all clocks and variables are reset and the accumulation-fire-rest cycle can start again.

Several bursting behaviours are described in [24]. Here we discuss only three of them, as all impossibility results depend on Property 8 and all the automata extensions are similar.

*Tonic Bursting* is the behaviour of a neuron producing a burst sequence as a response to a persistent and excitatory input sequence. *Phasic Bursting* is the behaviour of a neuron producing a burst as a consequence of a persistent excitatory input sequence and then remaining quiescent. Obviously the preceding behaviours require the ability of producing bursts.

*Bursting-then-Spiking* is the behaviour of a neuron producing a burst as response to a persistent excitatory input sequence and then producing a periodic output sequence. Such a behaviour, similarly to Phasic and Tonic Bursting, depends on the neuron ability of producing bursts. Moreover it requires inter-emission memory, in order to detect the beginning of a persistent sequence.

*Property 9* Neurons cannot exhibit the *Tonic Bursting, Phasic Bursting* and *Bursting-then-Spiking* behaviours.

*Proof* Follows from Property 8. □

**Spike Frequency Adaptation.** *Spike Frequency Adaptation* is the behaviour of a neuron producing a decreasing-frequency output sequence as a response to a persistent excitatory input sequence. In other words, the inter-emission time difference increases as the time elapses. This behaviour requires the neuron to have inter-emission memory as it should be able to keep track of the time elapsed since the beginning of the input sequence.

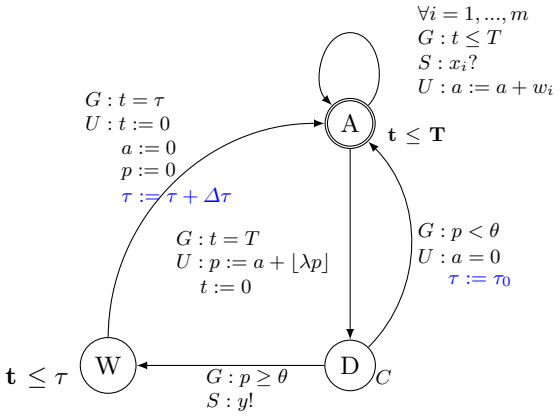



Fig. 5: The extended model for Spike Frequency Adaptation behaviour. Additions are colored in blue.

*Property 10* Neurons cannot reproduce the *Spike Frequency Adaptation* behaviour.

*Proof* The Spike Frequency Adaptation behaviour requires the neuron to detect the beginning of an excitatory input sequence and to increase the time required to fire a spike, after each emission. This means the neuron will produce different outcomes as response to equal inputs, which is impossible, as stated in Property 3. □

An extended neuron model able to reproduce Spike Frequency Adaptation behaviour is shown in Figure 5. This variant allows the refractory period to increase after each neuron emission, thus making the output frequency decrease.

**Spike Latency.** *Spike Latency* is the behaviour of a neuron firing delayed spikes, with respect to the instant when its potential reached or overcame the threshold. Such a delay is proportional to the strength of the signal which leads it to emission, i.e., the sum of weighed inputs received during the accumulation period preceding the emission. This behaviour requires the neuron to be able to postpone its output.

*Property 11* Neurons cannot reproduce the *Spike Latency* behaviour.

*Proof* The property holds by construction. As location **D** is *committed*, no firing can be delayed. □

An easy solution to extend our model is to introduce a delay between the instant the neuron reaches or overcomes its threshold and the actual emission instant. Such a delay $\delta$ depends on the sum of weighted inputs received during the last accumulation period. If the potential is greater than or equal to the threshold, the neuron computes the delay duration $\delta(a)$, assigning it to an integer variable $d$, and then waits in location **Del** for $d$ time units before emitting a spike on channel $y$. The extended version is depicted in Figure 6.

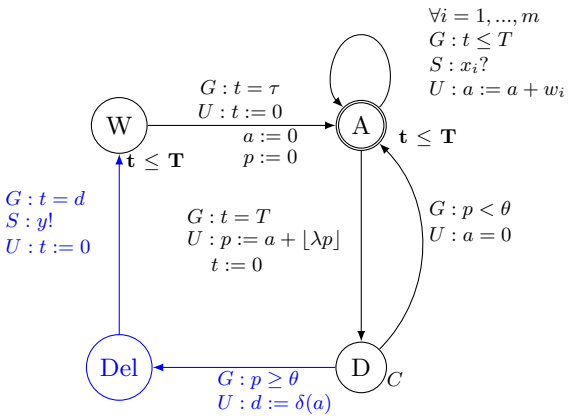



Fig. 6: The extended model for the Spike Latency behaviour. Additions are colored in blue.

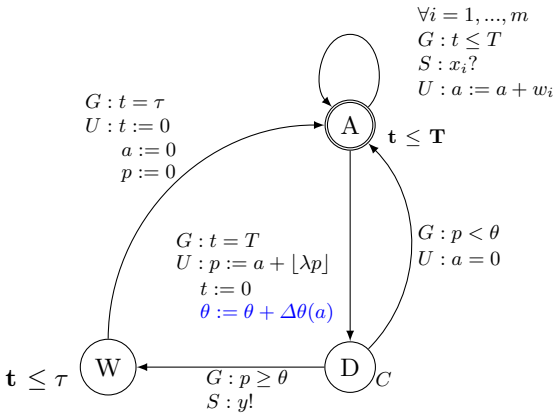



Fig. 7: The extended model for the Threshold variability behaviour. Additions colored are in blue.

**Threshold Variability.** *Threshold variability* is the behaviour of a neuron allowing its threshold to vary according to the strength of its inputs. More precisely, an excitatory input will rise the threshold while an inhibitory input will decrease it. As a consequence, excitatory inputs may more easily lead the neuron to fire when occurring after an inhibitory input.

*Property 12* Neurons cannot reproduce the *Threshold Variability* behaviour.

*Proof* By construction the neuron threshold never changes. □

The neuron model can be extended allowing the threshold to vary after each accumulation period according to the current sum of weighted inputs (see Figure

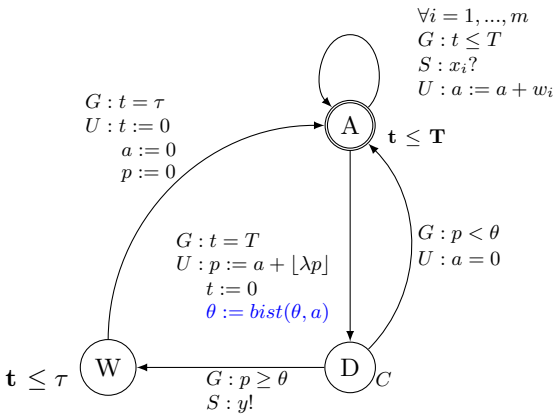



Fig. 8: The extended model for Bistability behaviour.
Additions are colored in blue.

7). The threshold variable initial value is $\theta_0$. On every firing of edge ($\mathbf{A} \rightarrow \mathbf{D}$), the threshold variable is increased of $\Delta(a)$, where $a$ is the sum of weighted inputs occurred during the last accumulation period and $\Delta(a)$ is an integer value whose sign is opposite to the sign of $a$ and whose magnitude is proportional to the magnitude of $a$.

**Bistability.** *Bistability* is the behaviour of a neuron alternating between two operation modes: *periodic emission* and *quiescence*. Upon reception of a single excitatory spike, it emits a periodic output sequence and switches to a quiescent mode (no emission) as soon as it received another spike. Such a behaviour requires the neuron to (i) be able to produce a periodic output sequence, even if no excitatory spike is received, (ii) be able to remain silent when no spike is received, and (iii) be able to switch between the two operation modes upon reception of an excitatory spike.

*Property 13* Neurons cannot reproduce the *Bistability* behaviour.

*Proof* The only possibility of obtaining a periodic output as a result of no excitatory input spike is to set $\theta = 0$. This is a limit case of Property 4. Since, by construction, the threshold cannot vary, the neuron cannot switch between the two operation modes. □

The neuron model can be modified as shown in Figure 8. This variant makes its threshold switch between 0 and a positive value at the end of any accumulation period during which it received an excitatory sum of weighted inputs $a$. A null threshold would make the neuron emit even if no input is received. Conversely, a positive threshold would prevent the neuron from emitting, if no input is received. Thus, on every firing of edge ($\mathbf{A} \rightarrow \mathbf{D}$), the threshold value $\theta$ is computed by the

function $bist(\cdot)$:

$$bist(\theta, a) = \begin{cases} 0 & if\ \theta > 0 \wedge a > 0 \\ \theta_0 & if\ \theta = 0 \wedge a > 0 \\ \theta & if\ a \leq 0. \end{cases}$$

**Inhibition-induced activities.** This is the behaviour of a neuron producing a spike output sequence as a response to a persistent *inhibitory* input sequence. We thus require the neuron to be able to emit as a consequence of some inhibitory input spikes.

*Property 14* Neurons cannot reproduce the *Inhibition-induced Spiking* behavior.

*Proof* Follows from Fact 1. □

An easy extension to our automata is to consider *the absolute value* of all inputs instead of their signed values.

**Rebound activities.** *Rebound Spike* is the behaviour of a neuron producing an output spike after it received an inhibitory input. Similarly to Inhibition-induced activities, this behaviour requires the neuron to emit as a consequence of an inhibitory input spike.

*Property 15* Neurons cannot exhibit the *Rebound Spiking* behaviour.

*Proof* Follows from Fact 1. □

We can modify our encoding by setting the neuron potential to be always non-negative and by fixing the threshold to be 0 as response to an inhibitory stimulation. Recall that a null threshold would make the neuron emit even if its potential is 0. Thus, on every firing of the edge $(\mathbf{A} \rightarrow \mathbf{D})$, if the current sum of weighted inputs $a$ is negative, the threshold $\theta$ is set to 0, otherwise it is set to a $\theta > 0$. This will allow an inhibitory stimulus to produce a rebound spike.

## 5 Parameter inference

In this section we examine the *Learning Problem*: i.e., how to determine a parameter assignment for a network with a fixed topology and a given input such that a desired output behaviour is displayed. Here we only focus on the estimation of synaptic weights in a given spiking neural network; the generalisation of our methodology to other parameters is left for future work.

Our analysis takes inspiration from the SpikeProp algorithm [5]; in a similar way, here, the learning process is led by *supervisors*. Differently from the previous section, each output neuron $\mathcal{N}$ is linked to a supervisor instead of an output consumer. Supervisors compare the expected output behaviour with the one of the output neuron they are connected to (function EVALUATE($\mathcal{N}$) in Algorithm 1). Thus either the output neuron behaved consistently or not. In the second case and in order to instruct the network, the supervisor back-propagates *advices* to the output neuron depending on two possible scenarios: i) the neuron fires a spike, but it was supposed to be quiescent, ii) the neuron remains quiescent, but it was

**Algorithm 1** The advice back-propagation algorithm

```
1: function ABP
2:     discovered = ∅
3:     for all 𝒩 ∈ Output do
4:         if 𝒩 ∉ discovered then
5:             discovered = discovered ∪ 𝒩
6:             if Evaluate(𝒩) = SHF then
7:                 SHF(𝒩)
8:             else if Evaluate(𝒩) = SNHF then
9:                 SNHF(𝒩)
```

**Algorithm 2** Should Have Fired algorithm

```
1: procedure Should-Have-Fired(𝒩)
2:     if 𝒩 ∈ discovered ∪ Output then
3:         return
4:     discovered = discovered ∪ 𝒩
5:     for all ℳ ∈ Pred(𝒩) do
6:         if ℳ ∉ Input then
7:             if Wt(𝒩, ℳ) ≥ 0 ∧ ¬ F(ℳ) then
8:                 SHF(ℳ)
9:             if Wt(𝒩, ℳ) < 0 ∧ F(ℳ) then
10:                SNHF(ℳ)
11:        Increase-Weight(𝒩, ℳ)
12:    return
```

supposed to fire a spike. In the first case the supervisor addresses a *should not have fired* message (SNHF) and in the second one a *should have fired* (SHF). Then each output neuron modifies its ingoing synaptic weights and in turn behaves as a supervisor with respect to its predecessors, back-propagating the proper advice.

The advice back-propagation (ABP), Algorithm 1, basically lies on a depth-first visit of the graph topology of the network. Let $\mathcal{N}_i$ be the $i$-th predecessor of an automaton $\mathcal{N}$, then we say that $\mathcal{N}_i$ fired, if it emitted a spike during the current or previous accumulate-fire-wait cycle of $\mathcal{N}$. Thus, upon reception of a SHF message, $\mathcal{N}$ has to *strengthen* the weight of each ingoing *excitatory* synapse and *weaken* the weight of each ingoing *inhibitory* synapse. Then, it propagates a SHF advice to each ingoing *excitatory* synapse (i.e., an arc with weight greater than 0: $\text{Wt} \geq 0$) corresponding to a neuron which *did not* fire recently ($\neg\text{F}(\mathcal{N})$ ), and symmetrically a SNHF advice to each ingoing *inhibitory* synapse ($\text{Wt} < 0$) corresponding to a neuron which fired recently (see Algorithm 2 for SHF, and Algorithm 3 for the dual case of SNHF). When the graph visit reaches an input generator, it will simply ignore any received advice (because input sequences should not be affected by the learning process). The learning process ends when all supervisors do not detect any more errors.

*Example 2 (Turning on and off a diamond structure of neurons.)* This example shows how the ABP algorithm can be used to make a neuron emit at least once in a spiking neural network having the *diamond* structure shown in Figure 9. We

**Algorithm 3** Abstract ABP: Should *Not* Have Fired advice pseudo-code

```
1:  procedure SHOULD-NOT-HAVE-FIRED(neuron)
2:      if 𝒩 ∈ discovered ∪ Output then
3:          return
4:      discovered = discovered ∪ 𝒩
5:      for all ℳ ∈ PREDECESSORS(𝒩) do
6:          if ℳ ∉ Input then
7:              if WT(𝒩, ℳ) ≥ 0 ∧ F(ℳ) then
8:                  SNHF(ℳ)
9:              if WT(𝒩, ℳ) < 0 ∧ ¬ F(ℳ) then
10:                 SHF(ℳ)
11:         DECREASE-WEIGHT(𝒩, ℳ)
12:     return
```

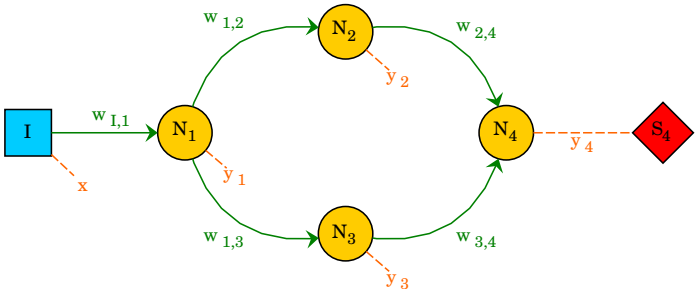



Fig. 9: A neural network with a diamond structure.

assume that $\mathcal{N}_1$ is fed by an input generator $\mathcal{I}$ that continuously emits spikes. No neuron in the network is able to emit because all the weights of their input synapses are equal to zero and their thresholds are higher than zero. We want the network to learn a weight assignment so that $\mathcal{N}_4$ is able to emit, that is, to produce a spike after an initial pause.

At the beginning we expect no activity from neuron $\mathcal{N}_4$. As soon as the initial pause is elapsed, we require a spike but, as all weights are equal to zero, no emission can happen. Thus a SHF advice is back-propagated to neurons $\mathcal{N}_2$ and $\mathcal{N}_3$ and consequently to $\mathcal{N}_1$. The process is then repeated until all weights stabilise and neuron $\mathcal{N}_4$ is able to fire. ◇

There are several possibilities on how to realise supervisors and the ABP algorithm. We propose here two approaches. The first one is model checking oriented and it is based on the idea that supervisors are represented by temporal logic formulae. The second one is simulation oriented, and the implementation of the algorithm is embedded into the timed automata modelling of the neuron.

**Model-checking-oriented approach.** Such a technique consists in iterating the learning process until a desired CTL property concerning the output of the

network is verified. The hypothesis we introduce are the following ones: (i) input generators, standard neurons, and output consumers share a global clock which is never reset and (ii) for each output consumer, there exists a clock measuring the elapsed time since the last received spike. The CTL formula specifying the expected output of the network can only contain predicates relative to the output consumers and the global clock. At each step of the algorithm, we make an external call to the model checker to test whether the network satisfies the formula or not. If the formula is verified, the learning process ends; otherwise, the model checker provides a trace as a counterexample. Such a trace is exploited to derive the proper corrective action to be applied to each output neuron, that is, the invocation of either the SHF procedure, or the SNHF procedure previously described (or no procedure).

More in detail, given a timed automata network representing some spiking neural network, we extend it with a global clock $t_g$ which is never reset and, for each output consumer $\mathcal{O}_K$ relative to the output neuron $\mathcal{N}_k$, we add a clock $s_k$ measuring the time elapsed since the last spike consumed by $\mathcal{O}_k$. Furthermore, let $state_{\mathcal{O}_k}(\mathbf{O})$ be an atomic proposition evaluating to true if the output consumer $\mathcal{O}_K$ is in its $\mathbf{O}$ location, and let $eval_{\mathcal{O}_k}(s_k)$ be an atomic proposition indicating the value of the clock $s_k$ in $\mathcal{O}_K$. In order to make it possible to deduce the proper corrective action, we impose the CTL formula describing the expected outcome of the network to be composed by the conjunction of sub-formulae respecting any of the patterns presented in the following.

Precise Firing. The output neuron $\mathcal{N}_k$ fires at time $t$:
$AF\,(\,t_g = t \wedge state_{\mathcal{O}_k}(\mathbf{O})\,)$.
The violation of such a formula requires the invocation of the SHF procedure.

Weak Quiescence. The output neuron $\mathcal{N}_k$ is quiescent at time $t$:
$AG\,(\,t_g = t \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,)$.
The SNHF procedure is called in case this formula is not satisfied.

Relaxed Firing. The output neuron $\mathcal{N}_k$ fires at least once within the time window $[\,t_1,\, t_2\,]$:
$AF\,(\,t_1 \leq t_g \leq t_2 \wedge state_{\mathcal{O}_k}(\mathbf{O})\,)$.
The violation of such a formula leads to the invocation of the SHF procedure.

Strong Quiescence. The output neuron $\mathcal{N}_k$ is quiescent for the whole duration of the time window $[\,t_1,\, t_2\,]$:
$AG\,(\,t_1 \leq t_g \leq t_2 \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,)$.
The SNHF procedure is needed in this case.

Precise Periodicity. The output neuron $\mathcal{N}_k$ eventually starts to periodically fire a spike with exact period $P$:
$AF(\,AG(\,eval_{\mathcal{O}_k}(s_k) \neq P \implies \neg state_{\mathcal{O}_k}(\mathbf{O})\,)$
$\wedge\, AG(\,state_{\mathcal{O}_k}(\mathbf{O}) \implies eval_{\mathcal{O}_k}(s_k) = P\,)\,)$.
If $\mathcal{N}_k$ fires a spike while the $s_k$ clock is different than $P$ or it does not fire a spike while the $s_k$ clock equals $P$, the formula is not satisfied. In the former (resp. latter) case, we deduce that the SNHF (resp. SHF) procedure is required.

Relaxed Periodicity. The output neuron $\mathcal{N}_k$ eventually begins to periodically fire a spike with a period that may vary in $[\,P_{min},\, P_{max}\,]$:
$AF(\,AG(\,eval_{\mathcal{O}_k}(s_k) \notin [\,P_{min},\, P_{max}\,] \implies$
$\neg state_{\mathcal{O}_k}(\mathbf{O})\,) \wedge$

$AF(\, state_{\mathcal{O}_k}(\mathbf{O}) \implies$
$P_{min} \leq eval_{\mathcal{O}_k}(s_k) \leq P_{max}\,)\,)$.
For the corrective actions, see the previous case.

As for future work, we intend to extend this set of CTL formulae with new formulae concerning the comparison of the output of two or more given neurons. Please notice that the Uppaal model-checker only supports a fragment of CTL where the use of nested path quantifiers is not allowed. Another model-checker should be called in order to fully exploit the expressive power of CTL.

Next we give a couple of examples.

*Example 3 (Diamond network.)* In this example, we apply the model-checking approach to the diamond network of Figure 9. Neuron $\mathcal{N}_4$ cannot emit a spike with the initial weights and parameters, which are:

| $w_{0,1}$ | $w_{1,2}$ | $w_{1,3}$ | $w_{2,4}$ | $w_{3,4}$ |
|---|---|---|---|---|
| 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |

| Neuron | T | $\theta$ | $\tau$ | $\lambda$ |
|---|---|---|---|---|
| $\mathcal{N}_1$ | 2 | 0.35 | 3 | 7/9 |
| $\mathcal{N}_2$ | 2 | 0.35 | 3 | 7/9 |
| $\mathcal{N}_3$ | 2 | 0.35 | 3 | 7/9 |
| $\mathcal{N}_4$ | 2 | 0.55 | 3 | 1/2 |

We expect $\mathcal{N}_4$ to spike every 20 time units. After three cycles of the algorithm (i.e., three checks of the formula and modifications of weights), we reach the following weight assignment:

| $w_{0,1}$ | $w_{1,2}$ | $w_{1,3}$ | $w_{2,4}$ | $w_{3,4}$ |
|---|---|---|---|---|
| 0.3 | 0.3 | 0.3 | 0.3 | 0.3 |

◇

*Example 4 (Mutual inhibition networks.)* In this example we focus on mutual inhibition networks, where the constituent neurons inhibit each other neuron's activity. These networks belong to the set of Control Path Generators (CPGs), which are known for their capability to produce rhythmic patterns of neural activity without receiving rhythmic inputs [22]. CPGs underlie many fundamental rhythmic activities such as digesting, breathing, and chewing. They are also crucial building blocks for the locomotor neural circuits both in invertebrate and vertebrate animals. It has been observed that, for suitable parameter values, mutual inhibition networks present a behaviour of the kind "winner takes all", that is, at a certain time one neuron becomes (and stays) activated and the other ones are inhibited [13].

We consider a mutual inhibition network of four neurons, as shown in Figure 10. This example, although being small, it is not trivial as it features inhibitor and excitatory edges as well as cycles.

We look for synaptical weights such that the "winner takes all" behaviour is displayed. We assume each neuron to be fed by an input generator $\mathcal{I}$ that continuously emits spikes. At the beginning, all the neurons have the same parameters (that is, firing threshold, remaining coefficient, accumulation period, and refractory period), and the weight of excitatory (resp. inhibitory) edges is set to 1 (resp.

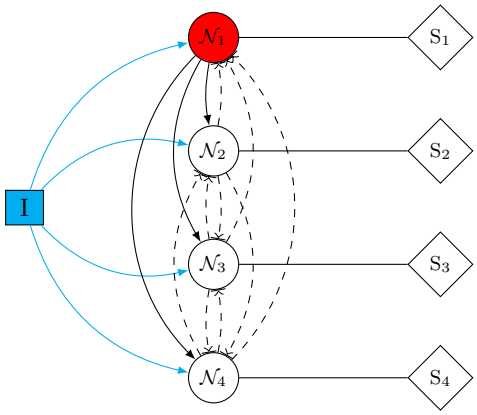



Fig. 10: We denote neurons by $\mathcal{N}_i$. The network is fed by an input generator $\mathcal{I}$ and the learning process is led by the supervisors $\mathcal{S}_i$. Dotted (resp. continuous) edges stand for inhibitions (resp. activations).

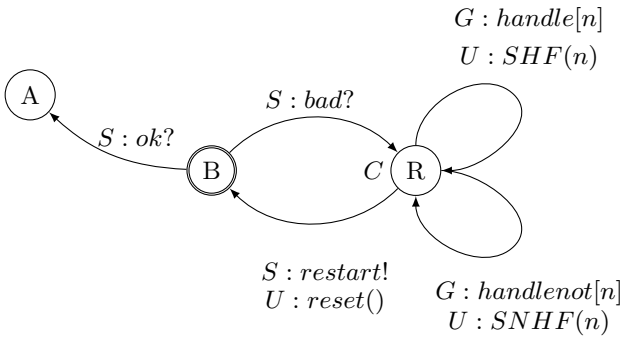



Fig. 11: The automata responsible of the $ABP - alg$

-1). We use the ABP algorithm to learn a weight assignment so that the first neuron is the winner. More precisely, we find a weight assignment so that, whatever the chosen path in the corresponding automata network is, the network stabilises when the global clock $t_g$ equals 70. The weight of the edges from the input generator $\mathcal{I}$ to the four neurons equals 0.041. The weight of the edges inhibiting $\mathcal{N}_1$ (resp. $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$) is -0.719 (resp. -0.817).

◇

**Simulation-oriented approach.** In this second approach, parameters are modified during the simulation of the network. This entails that the encoding of neurons needs to be adjusted in order to take care of the adaptation of such parameters. Algorithm ABP is realised by a dedicated automaton $ABP - alg$, that is depicted in Figure 11, and the role of supervisor is given to output consumers, that are modified as in Figure 12.

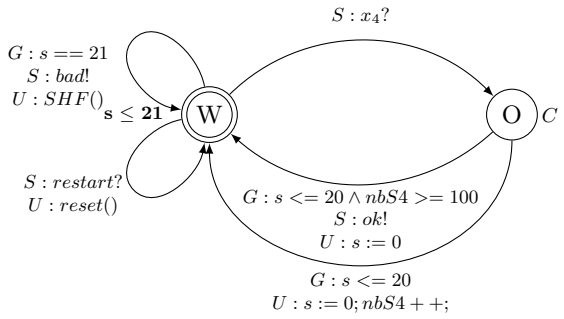



Fig. 12: Example of an output consumer in the simulation approach for the diamond structure

The idea is that, according to the function EVALUATE, if the corresponding output neuron misbehaves, then its output consumer sets whether it has to be treated according to the SHF or the SNHF function. Furthermore, it signals to the $ABP - alg$ through the message *bad*! that some adjustments on the network have to be done. Then the $ABP - alg$ automaton takes the lead and it recursively applies the function SHF or SNHF (this is achieved by setting a proper variable in a vector named *handle*) on the predecessors of the output neuron. Once there is no more neuron to whom the algorithm should be applied (for instance all neurons in the current run have been visited), the simulation is restarted in the network with the new parameters. If the output consumer does not recognise any misbehaviour, than it sends an *ok*! message to the $ABP - alg$ automaton, that in turn moves to an accepting state $A$.

More formally:

**Definition 9 (Output consumer for the simulation approach)** An output consumer for the simulation approach is a timed automaton

$$\mathcal{N} = (L, \mathbf{A}, X, Var, \Sigma, Arcs, Inv)$$

with:

- $L = \{\mathbf{W}, \mathbf{O}\}$ with **O** committed,
- $X = \{s\}$
- $Var = \{nb, handle[N], handlenot[N]\}$
- $\Sigma = \{x_i \mid x_i \text{ is an output neuron}\} \cup \{bad, restart, ok\}$,
- $Arcs = \{(\mathbf{W}, s < T_{min}, bad!, \{SHF()\}, \mathbf{W}),$
  $(\mathbf{W}, , restart?, \{s := 0, nb := 0\}, \mathbf{W}),$
  $(\mathbf{W}, , x_i?, , \mathbf{0}), (\mathbf{O}, s > T_{max}, bad!, \{SNHF()\}, \mathbf{W}),$
  $(\mathbf{O}, , restart?, \{s := 0, nb := 0\}, \mathbf{O}),$
  $(\mathbf{O}, good\_pattern, ok!, \{s := 0\}, \mathbf{W}),$
  $(\mathbf{O}, good\_not\_finished, , \{s := 0\}, \mathbf{W})\}$ ;
- $Inv(\mathbf{W}) = s \leq T$, $Inv(\mathbf{O}) = \mathbf{true}$

where the functions SHF and SNHF modify the global variables $handle_i$ and $handlenot_i$ respectively, that are used in the $ABP - alg$ automaton.

Notice that the precise definition (for instance the value of parameters $good\_pattern$, $good\_not\_finished$, $T_{min}$, $T_{max}$) of the output consumer depends on the expected behaviour of the supervisor. As an example, in Figure 12, that depicts the output consumer for the diamond network, we expect the output neuron to emit a spike within each window of 20 seconds for at least 100 times.

More in detail, the cycle from **W** to **W** is taken whenever a spike has not been sent before $T$ time units. Thus the automaton sends to the $ABP - alg$ automaton a *bad* message (signalling that an adjustment to the network should take place), and the update part handles the fact that we should perform the SHF algorithm (a variable corresponding to the concerned output neuron is set to **true** in the array *handle*). From the same state, whenever the message *restart* is received, all the variables of the automaton are reset to 0. When a spike from the output neuron $x$ is received, the output consumer moves to the state **O**. In this state, if the spike was received too late (and similarly as in the previous case), a *bad* message is sent to the $ABP - alg$. Otherwise, if everything was received on time and if the expected pattern has been completely verified, then an *ok* message is sent and the automaton moves to the initial state.

In the following, we give the formal definition of the $ABP - alg$ automaton:

**Definition 10 ($ABP - alg$ automaton)**

- $L = \{\mathbf{A}, \mathbf{B}, \mathbf{R}\}$ with **R** committed,
- $X = \emptyset$
- $Var = \{handle[N], handlenot[N]\}$
- $\Sigma = \{bad, ok, restart\}$,
- $Arcs = \{(\mathbf{B}, , ok?, , \mathbf{A}), (\mathbf{B}, , bad?, , \mathbf{R})$
  $(\mathbf{R}, handle[n], , SHF(n), \mathbf{R})$
  $(\mathbf{R}, handlenot[n], , SNHF(n), \mathbf{R})$
  $(\mathbf{R}, finished?(), restart!, reset(), \mathbf{B})\}$ ;
- $Inv(\mathbf{A}) = \mathbf{true}, Inv(\mathbf{B}) = \mathbf{true}, Inv(\mathbf{R}) = \mathbf{true}$.

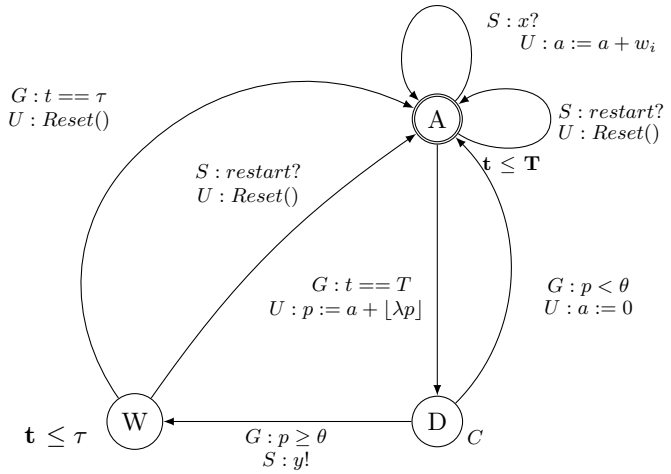
The arc from the state **B** to the accepting state **A** is taken whenever the Output consumer has finished the analysis of its pattern. Conversely, the arc from the state **B** to **R** is adopted when one of the output neurons has misbehaved (signalled by the reception of the message *bad*). In the committed state **R**, the parameters of the neural network are changed accordingly to the Algorithms 2 and 2. The arrays *handle* (respectively *handlenot*) have the information on the neurons to which the Algorithm for SHF (respectively SNHF) has to be applied. Once the cycle of updates finishes (checked through the function $finished?()$), the simulation is restarted by broadcasting a message *restart* to all the neurons and the output consumer.

Last, we give the definition of the changes induced in the standard neuron. As the update of the neuron parameters is done at the level of $ABP - alg$, the only change concerns the treatment of the signal *restart*. To this aim an arc handling the reception of the message is added to the states **W** and **A**.

**Definition 11 ( Standard Neuron for the simulation approach)** Given a neuron $v = (\theta, \tau, \lambda, p, y)$ with $m$ input synapses, its encoding into timed automata is $\mathcal{N} = (L, \mathbf{A}, X, Var, \Sigma, Arcs, Inv)$ with:

- $L = \{\mathbf{A}, \mathbf{W}, \mathbf{D}\}$ with **D** committed,

Fig. 13: Example of a neuron in the simulation approach for the diamond network




- $X = \{t\}$
- $Var = \{p, a\}$
- $\Sigma = \{x_i \mid i \in [1..m]\} \cup \{y\}$,
- $Arcs = \{(\mathbf{A}, t \leq T, x_i?, \{a := a + w_i\}, \mathbf{A}) \mid i \in [1..m]\} \cup \{(\mathbf{A}, t = T, \ , \{p := a + \lfloor \lambda p \rfloor\}, \mathbf{D}),$
  $(\mathbf{D}, p < \theta, \ , \{a := 0\}, \mathbf{A}), (\mathbf{D}, p \geq \theta, y!, \ , \mathbf{W}),$
  $(\mathbf{W}, t = \tau, \ , \{a := 0, t := 0, p := 0\}, \mathbf{A}),$
  $(\mathbf{A}, , restart?, \{a" = 0, t := 0, p := 0\}\mathbf{A}),$
  $(\mathbf{W}, , restart?, \{a" = 0, t := 0, p := 0\}\mathbf{A})\}$ ;
- $Inv(\mathbf{A}) = t \leq T, Inv(\mathbf{W}) = t \leq \tau, Inv(\mathbf{D}) = \mathbf{true}$.

Notice that the algorithm can be refined by setting some priorities on the neurons. For instance, if any additional information is known in advance on the behaviour of a specific neuron, its actions can be constrained and the corrective operations could be ignored. In some cases, the type of synapse (excitatory or inhibitory) can also be set, disallowing the possibility of changing its nature (e.g., from inhibitory to excitatory).

*Example 5 (Diamond network)* In this example, we apply the simulation-oriented approach on the diamond network on Figure 9. We take the same parameters and aim as in Example 3.

After the simulation, we obtain the following resulting weights :

| $w_{0,1}$ | $w_{1,2}$ | $w_{1,3}$ | $w_{2,4}$ | $w_{3,4}$ |
|---|---|---|---|---|
| 0.2 | 0.3 | 0.3 | 0.3 | 0.3 |

We obtain these weights when the global clock equals 42 and after only 2 sends of the message *bad* (meaning that the neuronal network is stabilising in only two applications of the ABP algorithm). Note that the resulting weights for the model-checking-oriented approach, seen in the precedent example, are different. Indeed, different sets of weights can give the same behaviour.

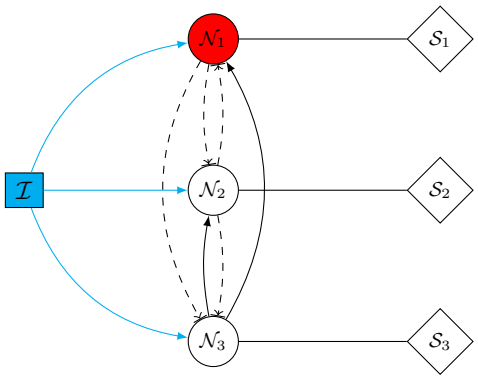



Fig. 14: A second neural network with "winner takes all" behaviour

◇

*Example 6 (Mutual inhibition network )* This example shows the weight estimation for the network of Figure 14 with the simulation approach. As before, we require a "winner takes all" behaviour where $\mathcal{N}_1$ is the winner. We expect that $\mathcal{N}_1$ spikes every 10 time units at most, and the neurons $\mathcal{N}_2$ and $\mathcal{N}_3$ do not spike. We take a network with the following initial parameters:

| Neuron | T | $\theta$ | $\tau$ | $\lambda$ |
|---|---|---|---|---|
| $\mathcal{N}_1$ | 2 | 0.75 | 3 | 1/2 |
| $\mathcal{N}_2$ | 2 | 0.75 | 3 | 1/2 |
| $\mathcal{N}_3$ | 2 | 0.75 | 3 | 1/2 |

| $w_{x,y}$ | $\mathcal{I}$ | $\mathcal{N}_1$ | $\mathcal{N}_2$ | $\mathcal{N}_3$ |
|---|---|---|---|---|
| $\mathcal{I}$ | 0 | 0.1 | 0.1 | 0.1 |
| $\mathcal{N}_1$ | 0 | 0 | -0.1 | -0.1 |
| $\mathcal{N}_2$ | 0 | -0.1 | 0 | -0.1 |
| $\mathcal{N}_3$ | 0 | 0.1 | 0.1 | 0 |

And we obtain the following weights in 5 executions of the ABP algorithm:

| $w_{x,y}$ | $\mathcal{I}$ | $\mathcal{N}_1$ | $\mathcal{N}_2$ | $\mathcal{N}_3$ |
|---|---|---|---|---|
| $\mathcal{I}$ | 0 | 0.6 | 0.1 | 0.1 |
| $\mathcal{N}_1$ | 0 | 0 | -0.1 | -0.1 |
| $\mathcal{N}_2$ | 0 | -0.1 | 0 | -0.1 |
| $\mathcal{N}_3$ | 0 | 0.6 | 0.1 | 0 |

◇

Notice that, as the application of the ABP algorithm in the simulation approach is non-deterministic, the parameters we found may depend on the precise execution and several solutions are therefore possible. The complete encoding of the examples shown here can be found at [12].

## 6 Related Work

To the best of our knowledge, there are few attempts of giving formal models for LI&F. Apart from the already discussed approach of [13], where the authors model and verify LI&F networks thanks to the synchronous language Lustre, the closest related work we are aware of is [3]. In this work, the authors propose a mapping of spiking neural P systems into timed automata. The modelling is substantially different from ours. They consider neurons as static objects and the dynamics is given in terms of evolution rules while for us the dynamics is intrinsic to the modelling of the neuron. This, for instance, entails that inhibitions are not just negative weights as in our case, but are represented as *forgetting rules*. On top of this, the notion of time is also different: while they consider durations in terms of number of applied rules, we have an explicit notion of duration given in terms of accumulation and refractory period.

As far as our parameter learning approach is concerned, we borrow inspiration from the SpikeProp rule [5], a variant of the well known back-propagation algorithm [32] used for supervised learning in second generation learning. The SpikeProp rule deals with multi-layered cycle-free spiking neural networks and aims at training networks to produce a given output sequence for each class of input sequences. The main difference with respect to our approach is that we are considering here a discrete model and our networks are not multi-layered. We also rest on Hebb's learning rule [19] and its time-dependent generalisation rule, the spike timing dependent plasticity (STDP) rule [33], which aims at adjusting the synaptical weights of a network according to the time occurrences of input and output spikes of neurons. It acts locally, with respect to each neuron, i.e., no prior assumption on the network topology is required in order to compute the weight variations for some neuron input synapses. Differently from the STDP, our approach takes into account not only recent spikes but also some external feedback (*advices*) in order to determine which weights should be modified and whether they must increase or decrease. Moreover, we do not prevent excitatory synapses from becoming inhibitory (or vice versa), which is usually a constraint for STDP implementations. A general overview on spiking neural network learning approaches and open problems in this context can be found in [18].

## 7 Conclusion

In this paper we formalised the LI&F model of spiking neural networks via timed automata networks. We have a complete implementation of the proposed model and examples via the tool `Uppaal`, that can be found at the pages [6] and [12]. In our modeling framework, information processing is based on the precise timing of spike emissions rather than the average numbers of spikes in a given time window. Timed automata turned out to be very suited to model spiking neural networks, allowing us to take into account time-related aspects, such as the exact spike occurrence times and the refractory period, a lapse of time immediately following each spike emission, when the neuron emission capability is reduced.

In this work, we exploited model checking to automatically validate our automaton-based mapping of the LI&F model according to a number of behaviours (i.e., typical responses to an input pattern) the LI&F model should be able to reproduce,

namely tonic spiking, excitability, and integrator. Formal methods of computer science turned out to be an effective tool to validate our modelling approach. As for future work concerning the modeling aspects, we plan to provide analogous formalisations for more complex spiking neuron models, such as the theta-neuron model [14] or the Izhikevich one [23]. We also intend to extend our model to include propagation delays, which are considered important within the scope of spiking neural networks [29]. Our extension is intended to add suitable states and clocks to model synapses. We also plan to perform a robustness analysis of the obtained model, in order to detect which neuron parameters influence most the verification of some wished temporal properties.

As key contribution, we proposed a novel technique to infer the synaptical weights of spiking neural networks. At this aim, we adapted machine learning techniques to bio-inspired models, which makes our work original and complementary with respect to the main international projects aiming at understanding the human brain, such as the Human Brain Project [9], which mainly relies on large-scale simulations.

For our learning approach, we have focussed on a simplified type of supervisors: each supervisor describes the output of a single neuron in isolation from the other ones. Nonetheless, notice that the back-propagation algorithm is still valid for more complex scenarios that specify and compare the behaviour of groups of neurons. As for future work, we intend to formalise more sophisticated supervisors, allowing to compare the output of several neurons. Moreover, to refine our learning algorithm, we could exploit results coming from the gene regulatory network domain, where a link between the topology of the network and its dynamical behaviour is established [31].

As a last step, we plan to generalise our technique in order to be able to infer not only synaptical weights but also other parameters, such as the leak factor or the firing threshold.

**Acknowledgements** We are grateful to Giovanni Ciatto for his preliminary implementation work and for his enthusiasm in collaborating with us.

## References

1. Ackley, D.H., Hinton, G.E., Sejnowski, T.J.: A learning algorithm for boltzmann machines. In: D. Waltz, J.A. Feldman (eds.) Connectionist Models and Their Implications: Readings from Cognitive Science, pp. 285–307. Ablex Publishing Corp., Norwood, NJ, USA (1988)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Aman, B., Ciobanu, G.: Modelling and verification of weighted spiking neural systems. Theoretical Computer Science **623**, 92 – 102 (2016). DOI http://dx.doi.org/10.1016/j.tcs.2015.11.005. URL http://www.sciencedirect.com/science/article/pii/S0304397515009792
4. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In: Proceedings of Workshop on Verification and Control of Hybrid Systems III, no. 1066 in Lecture Notes in Computer Science, pp. 232–243. Springer–Verlag (1995)
5. Bohte, S.M., Poutré, H.A.L., Kok, J.N., La, H.A., Joost, P., Kok, N.: Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing **48**, 17–37 (2002)
6. Ciatto, G., De Maria, E., Di Giusto, C.: Additional material. https://github.com/gciatto/snn_as_ta (2016)

7. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (1999)
8. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems **2**(4), 303–314 (1989)
9. D'Angelo, E., Danese, G., Florimbi, G., Leporati, F., Majani, A., Masoli, S., Solinas, S., Torti, E.: 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, pp. 740–747. IEEE Computer Society (2015). DOI 10.1109/DSD.2015.80. URL https://doi.org/10.1109/DSD.2015.80
10. De Maria, E., Di Giusto, C.: Parameter learning for spiking neural networks modelled as timed automata. In: P. Anderson, H. Gamboa, A.L.N. Fred, S.B. i Badia (eds.) 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., pp. 17–28. SciTePress (2018). DOI 10.5220/0006530300170028. URL https://doi.org/10.5220/0006530300170028
11. De Maria, E., Di Giusto, C., Ciatto, G.: Formal validation of neural networks as timed automata. In: Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics, Nha Trang City, Viet Nam, December 7-8, 2017, pp. 15–22. ACM (2017). DOI 10.1145/3156346.3156350. URL http://doi.acm.org/10.1145/3156346.3156350
12. De Maria, E., Di Giusto, C., Laversa, L.: Additional material. https://digiusto.bitbucket.io/ (2017)
13. De Maria, E., Muzy, A., Gaffé, D., Ressouche, A., Grammont, F.: Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models. In: Fifth International Workshop on Hybrid Systems Biology. Grenoble, France (2016)
14. Ermentrout, G.B., Kopell, N.: Parabolic bursting in an excitable system coupled with a slow oscillation. SIAM Journal on Applied Mathematics **46**(2), 233–253 (1986)
15. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Machine Learning **37**(3), 277–296 (1999)
16. Gerstner, W., Kistler, W.: Spiking Neuron Models: An Introduction. Cambridge University Press, New York, NY, USA (2002)
17. Gruber, H., Holzer, M., Kiehn, A., König, B.: On timed automata with discrete time - structural and language theoretical characterization. In: Developments in Language Theory, 9th International Conference, DLT 2005, Palermo, Italy, July 4-8, 2005, Proceedings, pp. 272–283 (2005). DOI 10.1007/11505877_24
18. Grüning, A., Bohte, S.: Spiking neural networks: Principles and challenges (2014)
19. Hebb, D.O.: The Organization of Behavior. John Wiley (1949)
20. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. The Journal of Physiology **117**(4), 500–544 (1952)
21. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. In: J.A. Anderson, E. Rosenfeld (eds.) Neurocomputing: Foundations of Research, pp. 457–464. MIT Press, Cambridge, MA, USA (1988)
22. Ijspeert, A.J.: Central pattern generators for locomotion control in animals and robots: A review. Neural Networks **21**(4), 642–653 (2008). DOI 10.1016/j.neunet.2008.03.014. URL http://dx.doi.org/10.1016/j.neunet.2008.03.014
23. Izhikevich, E.M.: Simple model of spiking neurons. Trans. Neur. Netw. **14**(6), 1569–1572 (2003)
24. Izhikevich, E.M.: Which model to use for cortical spiking neurons? IEEE Transactions on Neural Networks **15**(5), 1063–1070 (2004)
25. Lapicque, L.: Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. J Physiol Pathol Gen **9**, 620–635 (1907)
26. Maass, W.: Networks of spiking neurons: The third generation of neural network models. Neural Networks **10**(9), 1659 – 1671 (1997)
27. Matsuoka, K.: Mechanisms of frequency and pattern control in the neural rhythm generators. Biological cybernetics **56**(5-6), 345–353 (1987)
28. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics **5**(4), 115–133 (1943)
29. Paugam-Moisy, H., Bohte, S.: Computing with Spiking Neuron Networks, pp. 335–376. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

30. Recce, M.: Encoding information in neuronal activity. In: W. Maass, C.M. Bishop (eds.) Pulsed Neural Networks, pp. 111–131. MIT Press, Cambridge, MA, USA (1999)
31. Richard, A.: Negative circuits and sustained oscillations in asynchronous automata networks. Advances in Applied Mathematics **44**(4), 378–392 (2010)
32. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. In: J.A. Anderson, E. Rosenfeld (eds.) Neurocomputing: Foundations of Research, pp. 696–699. MIT Press, Cambridge, MA, USA (1988)
33. Sjöström, J., Gerstner, W.: Spike-timing dependent plasticity. Scholarpedia **5**(2) (2010)