Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models

Elisabetta de Maria, Alexandre Muzy, Daniel Gaffé, Annie Ressouche, Franck Grammont

▶ **To cite this version:**

Elisabetta de Maria, Alexandre Muzy, Daniel Gaffé, Annie Ressouche, Franck Grammont. Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models. [Research Report] RR-8937, UCA, Inria; UCA, I3S; UCA, LEAT; UCA, LJAD. 2016, pp.21. hal-01349019

HAL Id: hal-01349019
https://inria.hal.science/hal-01349019

Submitted on 26 Jul 2016

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

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

Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models

Elisabetta de Maria , Alexandre Muzy , Daniel Gaffé , Annie Ressouche , Franck Grammont

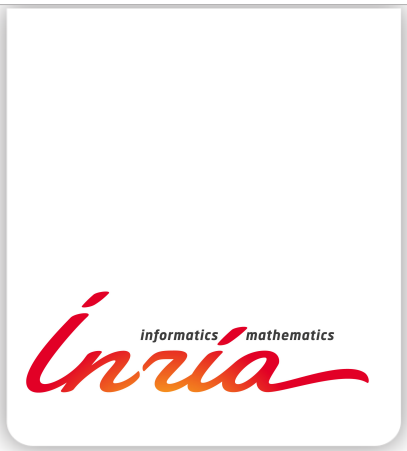
RESEARCH REPORT

N° 8937

August 2016

Project-Team Stars

ISSN 0249-6399 ISRN INRIA/RR--8937--FR+ENG



Verification of Temporal Properties of Neuronal Archetypes Using Synchronous Models

Elisabetta de Maria *, Alexandre Muzy †, Daniel Gaffé ‡, Annie Ressouche §, Franck Grammont¶

Project-Team Stars

Research Report n° 8937 — August 2016 — 18 pages

Abstract: There exists many ways to connect two, three or more neurons together to form different graphs. We call archetypes only the graphs whose properties can be associated with specific classes of biologically relevant structures and behaviors. These archetypes are supposed to be the basis of typical instances of neuronal information processing. To model different representative archetypes and express their temporal properties, we use a synchronous programming language dedicated to reactive systems (Lustre). The properties are then automatically validated thanks to several model checkers supporting data types. The respective results are compared and depend on their underlying abstraction methods.

Key-words: Neuronal archetypes, reactive systems, synchronous languages, temporal properties, model checking

* Université Côte d'Azur, CNRS, I3S, France, email: edemaria@i3s.unice.fr
† Université Côte d'Azur, CNRS, I3S, France, email: muzy@i3s.unice.fr
‡ Université Côte d'Azur, CNRS, LEAT, France, email: Daniel.Gaffe@unice.fr
§ Université Côte d'Azur, Inria, France, email: annie.ressouche@inria.fr
¶ Université Côte d'Azur, CNRS, LJAD, France, email: grammont@unice.fr

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Vérification de propriétés temporelles d'archétypes de neurones à l'aide de modèles synchrones

Résumé : Il existe maintes façons de connecter deux, trois neurones ou d'avantages ensemble pour former des graphes différents. Nous appellerons archétypes uniquement ceux dont les propriétés peuvent être associées à des classes spécifiques de structures et comportements significatifs biologiquement. Nous supposerons que ces archétypes sont les bases d'instances typiques du traitement de l'information neuronale. Pour modéliser différents archétypes représentatifs et exprimer leurs propriétés temporelles, nous utiliserons un langage de programmation synchrone dédié aux systèmes réactifs (Lustre). Ensuite, les propriétés seront validées automatiquement grâce à plusieurs model checkers qui supportent les types de données. Les résultats respectifs seront comparés et dépenderont des méthodes d'abstraction sous-jacentes.

Mots-clés : Archétypes neuronaux, systèmes réactifs, langages synchrones, propriétés temporelles, model checking

Contents

1 Introduction **3**

2 Synchronous Reactive Neuron Model **5**

3 A quick introduction to Lustre **7**

4 Encoding Neuronal Archetypes in Lustre **9**

5 Encoding and Verifying Temporal Properties of Archetypes in Lustre **11**
5.1 Simple Series (see Fig. 1(a)) 11
5.2 Series with Multiple Outputs (see Fig. 1(b)) 13
5.3 Parallel Composition (see Fig. 1(c)) 14
5.4 Negative Loop (see Fig. 1(d)) 14
5.5 Inhibition of a Behavior (see Fig. 1(e)) 15
5.6 Contralateral Inhibition (see Fig. 1(f)) 15
5.7 Comparison of the Model Checkers 17

6 Discussion and Future Work **17**

1 Introduction

Since a few years, the investigation of neuronal micro-circuits has become an emerging question in Neuroscience, notably in the perspective of their integration with neurocomputing approaches [14]. We call archetypes specific graphs of a few neurons with biologically relevant structures and behaviors. These archetypes correspond to elementary and fundamental elements of neuronal information processing. Several archetypes can be coupled to constitute the elementary bricks of bigger neuronal circuits in charge of specific functions. For instance, locomotive motion and other rhythmic behaviors are controlled by well-known specific neuronal circuits called Central Generator Pattern (CPG) [15]. These CPG have the capacity to generate oscillatory activities, at various regimes, thanks to some specific properties at the neuronal and circuit levels.

The goal of this work is to formally study the behavior of different representative archetypes. At this aim, we model the archetypes using a synchronous language for the description of reactive systems (Lustre). Each archetype (and corresponding assumed behavior in terms of neuronal information processing) is validated thanks to model checkers.

Different approaches have been proposed in the literature to model neural networks (Artificial Neural Networks [4], Spiking Neural Networks [12], etc.). In this paper we focus on Boolean Spiking Neural Networks where the neurons electrical properties are described via an integrate-and-fire model [6]. Notice that discrete modeling is well suited because neuronal activity, as with any recorded physical event, is only known through discrete recording (the recording sampling rate is usually set at a significantly higher resolution than the one of the recorded system, so that there is no loss of information). We describe neural networks as weighted directed graphs whose nodes represent neurons and whose edges stand for synaptic connections. At each time unit, all the neurons compute their membrane potential accounting not only for the current input signals but also for the ones received along a given temporal window. Each neuron can emit a spike according to the overtaking of a given threshold. Such a modeling is more sophisticated than the one proposed by McCulloch and Pitts in [16], where the behavior of a neural network is expressed

in terms of propositional logic and the present activity of each neuron does not depend on past events.

Spiking neural networks can be considered as reactive systems: their inputs are physiological signals coming from input synapses, and their outputs represent the signals emitted in reaction. This class of systems fits well with the synchronous approach based on the notion of a *logical time*: time is considered as a sequence of logical discrete *instants*. An instant is a point in time where external input events can be observed, along with the internal events that are a consequence of the latter. The synchronous paradigm can be implemented using synchronous programming languages. In this approach we can model an activity according to a logical time framing: the activity is characterized by a set of events expected at each logical instant and by their expected consequences. A synchronous system evolves only at these instants and is "frozen" otherwise (nothing changes between instants). At each logical instant, all events are instantaneously broadcasted, if necessary, to all parts of the system whose instantaneous reaction to these events contributes to the global system state. Synchronous programming languages being initially dedicated to digital circuits, this neural implementation could be easily mapped into a physical one.

Each instant is triggered by input events (the core information completed with the internal state computed from instantaneous broadcast performed during the instant frame). As a consequence, inputs and resulting outputs all occur simultaneously. This (ideal) *synchrony hypothesis* is the main characteristics of the synchronous paradigm [8]. Another major feature is also that it supports concurrency through a deterministic parallel composition. The synchronous paradigm is now well established relying on a rigorous semantics and on tools for simulation and verification.

Several synchronous languages respect this synchronous paradigm. All these languages have a similar expressivity. However, we choose here Lustre [8] *synchronous language* to express neuron behaviors more easily. Lustre defines operator networks interconnected with data flows and it is particularly well suited to express neuron networks. Lustre respects the *synchrony* hypothesis which divides time into discrete instants. It is a data flow language offering two main advantages: (1) it is *functional* with no complex side effects, making it well adapted to formal verification and safe program transformation; also, reuse is made easier, which is an interesting feature for reliable programming concerns; (2) it is a *parallel* model, where any sequencing and synchronization depends on data dependencies. Moreover, the Lustre formalism is close to temporal logic and this allows the language to be used for both writing programs and expressing properties as observers. Hence, Lustre offers an original verification means to prove that, as long as the environment behaves properly (i.e., satisfies some *assumption*), the program satisfies a given *property*. If we consider only safety properties, both the assumption and the property can be expressed by some programs, called *synchronous observers* [9]. An observer of a safety property is a program, taking as inputs the inputs/outputs of the program under verification, and deciding (e.g., by emitting an alarm signal) at each instant whether the property is violated. Running in parallel with the program, an observer of the desired property and an observer of the assumption made about the environment have just to check that either the alarm signal is never emitted (property satisfied) or the alarm signal is emitted (property violated). This can be done by a simple traversal of the reachable states of the compound program.

There exists several model checkers for Lustre that are well suited to our purpose: *Lesar* [10], *Nbac* [11], *Luke* [1], *Rantanplan* [5] and *kind2* [7]. Verification with Lesar is performed on an abstract (finite) model of the program. Concretely, for purely logical systems the proof is complete, whereas in general (in particular when numerical values are involved) the proof can be only partial. Indeed, properties related to values depend on the abstraction performed by the tool. In our experiment, some properties can be validated with Lesar, but some others need

powerful abstraction techniques. Hence, we use Lustre tools such as Nbac, Luke, Rantanplan and kind2. To perform abstractions, Lesar and NBac use convex polyhedra [13] representation of integers and reals. On the other hand, Luke is also another k-induction model checker, however it is based on propositional logic. Finally, Rantanplan and kind2 rely on SMT (Satisfiabitily Modulo Theories) based k-induction. kind2 has been specifically developed to prove safety properties of Lustre models, it combines several resolution engines and it turns out that it is the most powerful model checker used in this paper. This overall approach is used here to verify temporal properties of archetypes using model-checking techniques.

The paper is organized as follows. In Sect. 3 we present the computational model we adopt and in Sect. 4 we briefly introduce Lustre. In Sect. 4 we introduce the basic archetypes (series, series with multiple outputs, parallel composition, negative loop, inhibition of a behavior, contralateral inhibition) and we show how they can be modeled using Lustre. More precisely, we illustrate how the behavior of a single neuron can be encoded in a Lustre node and how two or more neurons can be connected to form a circuit. In Sect. 5 we express in Lustre important temporal properties concerning the described archetypes and we verify the satisfaction of these properties using the above-mentioned model checkers. Finally, Sect. 6 is devoted to a final discussion on the obtained results and on the future work.

2 Synchronous Reactive Neuron Model

We refer here to *synchronous reactive systems* as systems reacting under the synchronous assumption, i.e., as computing their states and sending instantaneously their output events when receiving input events. Synchronous reactive systems can be conceived as an abstraction of digital circuits. Therefore, to fit electronic/computational discreteness and finiteness, some assumptions according to the synchronous paradigm will be introduced.

We describe here first the structure of a neuron network as a graph. The dynamics of usual leaky integrate-and-fire spiking networks is presented later.

Definition 1. A *network of synchronous reactive neurons* is a *weighted directed graph* (G, w), where $G = (N, A)$ is a *directed graph* with $N = \{1, 2, \ldots, n\}$ the *set of neuron indexes* and $A = \{(i, j) \,|\, i, j \in N\}$ the *set of ordered pairs of neuron indexes (synapses)*, and $w : A \to \mathbb{Q}$ is the *synapse weight function*.

In a leaky integrate-and-fire neuron, the *membrane potential* of the neuron integrates the values of the action potentials received from its input neurons.

Definition 2. A *usual leaky integrate-and-fire model* is a structure $LIF_i = (I_i, Y_i, S_i, T_i, \Delta_i, \Lambda_i)$, where $I_i = \mathbb{B}$ is the *input alphabet*; $Y_i = \mathbb{B}$ is the *output alphabet*; $S_i = \mathbb{R}$ is the *set of states* defined as the set of values of *membrane potential*; $T_i = \mathbb{R}_0^+ \cup \{+\infty\}$ is the *time base*; $\Delta_i : I_i^m \times S_i \times T_i \to S_i$ is the *transition function* defined as

$$p_i' = \Delta_i(x_{i_1}, \ldots, x_{i_m}, p_i, t_i) = \begin{cases} \Sigma_{j \in Pred(i)} w_{ji} x_j & if\ p_i \geq \tau_i \\ r_i(t_i) p_i + \Sigma_{j \in Pred(i)} w_{ji} x_j & otherwise \end{cases}$$

where $Pred(i)$ is the *set of m predecessors* of neuron $i \in N$, x_j is the input of neuron $i \in N$ received from neuron $j \in Pred(i)$, $w_{ji} = w(j, i) \in \mathbb{Q}$ is the *synapse weight* from neuron $j \in N$ to neuron $i \in N$, $r_i(t_i)$ is the *remaining potential coefficient* (a decreasing function in time, usually $r_i(t_i) = exp(-\alpha t_i)$, with α a positive constant), and $\tau_i \in \mathbb{R}_0^+$ is the *firing threshold*, and $\Lambda_i : S_i \to Y_i$ is the *output function* defined as $\Lambda_i(p_i) = y_i = \begin{cases} 1 & if\ p_i \geq \tau_i \\ 0 & otherwise \end{cases}$.

For each synapse $(j,i) \in A$ between a neuron $j \in N$ and a neuron $i \in N$, $y_i \in \mathbb{B}$ is the *output spike value* emitted by neuron i, and $x_j \in \mathbb{B}$ is the *input spike value* of neuron i received from neuron j. If the membrane potential p_i is above the threshold τ_i, at the next transition the output spike value is set to $y_i = 1$ and the membrane potential is reset to $p_i = 0$. When the remaining potential coefficient r_i is a constant equal to 1, there is *no leak*, all the potential received at last transition remains in the neuron soma. When $r_i = 0$, all the potential received at last transition is lost (the model is then equivalent to McCulloch & Pitts' model [16]).

The usual leaky integrate-and-fire model presented in Definition 2 is not compatible with our synchronous reactive system assumption: both state and time sets are possibly infinite (cf. $r_i(t_i) = exp(-\alpha t_i)$, the exponentially decreasing function defined for $t_i \in [0, +\infty]$). We will show now how to approximate and limit potential values to fit the synchronous reactive system assumption.

The membrane potential can be defined as an integral of previous input values received by the neuron. This integral can be approximated by a sum and a power law leading to $p(t) = \Sigma_{e=0}^{+\infty}\Sigma_{j=1}^{m} r^e x_j(t-e)$[1], where $e \in \mathbb{R}_0^+ \cup \{+\infty\}$ represents the *time elapsed* until the current time t. The membrane potential *integrates* both current input values and what remains from previous inputs. As remaining input potentials decrease with time following a power law, inputs received a long time ago can nevertheless be neglected. Only remaining input potentials greater than a *threshold error* ϵ can be considered, *i.e.*, $r^e \geq \epsilon$. Thus only elapsed times $e \leq \frac{ln(\epsilon)}{ln(r)}$ can be taken into account, where $\sigma = \frac{ln(\epsilon)}{ln(r)}$ is the *integration time window*, *i.e.*, the period over which the neuron integrates past input values. For example, an error $\epsilon = 1\%$ and a remaining coefficient $r = 50\%$ correspond to an integration window $\sigma = 6.64$. This means that, *sliding the integration window* of a width equal to σ, at each time t, no input older that $e = 6.64$ will be considered, leading to an error of $\epsilon = 1\%$ in the membrane potential. The *time-dependence of the membrane potential is not anymore infinite but bounded to* $[t-\sigma, t]$. State changes need now to be finite. If we discretize each time step with $t, e \in \mathbb{N}_0$ (leading to $\sigma = \lceil\sigma\rceil = \lceil 6.64\rceil = 7$ for the previous example), the membrane potential $p(t)$ consists of a sum $\Sigma_{j=1}^{m} x_j(t) + r\Sigma_{j=1}^{m} x_j(t-1) + r^2\Sigma_{j=1}^{m} x_j(t-2) + r^3\Sigma_{j=1}^{m} x_j(t-3) + \ldots + r^\sigma\Sigma_{j=1}^{m} x_j(t-\sigma)$, *i.e.*, $p(t) = \Sigma_{e=0}^{\sigma} r^e \Sigma_{j=1}^{m} x_j(t-e)$.

Thanks to the previous time boundness and discreteness, each time t can be considered as a particular *transition*. The *computation of the membrane potential now depends on a finite memory of maximum size* $m \times (\sigma + 1)$, with $m, \sigma \in \mathbb{N}_0$.

A last simplification of the usual leaky integrate-and-fire model presented in Definition 2 concerns the real values of the membrane potential. Indeed, real numbers are approximated by computers as floating-point values. However, rational numbers are needed to get efficient results from model checkers. In our case, notice that this assumption well fits with remaining coefficients (that can easily be approximated by Taylor series or simple percentages).

Accounting for all the previous assumptions on usual leaky integrate-and-fire neurons, the following definition can be provided for their mapping to synchronous reactive systems (finite state automata are proved to be equivalent to synchronous programs as Lustre or Esterel [3]).

Definition 3. A *synchronous reactive neuron* implements a *leaky integrate and fire model* as a *finite state machine (FSM)* $SRN_i = (I_i, Y_i, S_i, \Delta_i, \Lambda_i)$, where $I_i = \mathbb{B}$ is the *input alphabet*; $Y_i = \mathbb{B}$ is the *output alphabet*; $S_i = \mathbb{Q}$ is the *set of membrane potential values*; $\Delta_i : I_i^{m\times(\sigma_i+1)} \rightarrow S_i$ is the *transition function* defined as $p_{i_k} = \Delta_i(\mathbf{X_k}) = \mathbf{W}\,\mathbf{X_k}\,\mathbf{R}$, where

[1] For the sake of simplicity, we assume that all the synaptic weights are equal to 1; supposing that neuron i has m predecessors, we write $\Sigma_{j=1}^{m} x_j$ to denote $\Sigma_{j\in Pred(i)} x_j$; when there is no ambiguity on the neuron index, we do not indicate it.

- $\mathbf{W} = [w_1, w_2, \ldots, w_m]$ is the *vector of synaptic weights* (each column corresponds to an input $j \in \{1, \ldots, m\}$ and $w_j \in \mathbb{Q}$),
- $\mathbf{X_k} = \begin{bmatrix} x_{10} & x_{11} & \cdots & x_{1\sigma_i} \\ x_{20} & x_{21} & \cdots & x_{2\sigma_i} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m0} & x_{m1} & \cdots & x_{m\sigma_i} \end{bmatrix}_k$ is a *matrix of Boolean stored input values* (each row corresponds to an input $j \in \{1, \ldots, m\}$ and each column to an *elapsed time* $e_i \in \{0, \ldots, \sigma_i\}$ with $\sigma_i \in \mathbb{N}_0$),
- $\mathbf{R} = \begin{bmatrix} 1 \\ r_i \\ \vdots \\ r_i^{\sigma_i} \end{bmatrix}$ is the *vector of remaining coefficients* with $r_i \in \mathbb{Q}$, and
- $\tau_i \in \mathbb{Q}$ is the *firing threshold*.

In particular $\mathbf{X_0} = \begin{bmatrix} I_1 & 0 & \cdots & 0 \\ I_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ I_{m0} & 0 & \cdots & 0 \end{bmatrix}$ and $\mathbf{X_k} = \begin{cases} shift([\mathbf{0}], I_k) & if\ p_{i_{k-1}} \geq \tau_i \\ shift(\mathbf{X_{k-1}}, I_k) & otherwise \end{cases}$

where $shift$ is a shifting matrix function: $M^{m \times (\sigma_i + 1)} \times V^m \rightarrow M^{m \times (\sigma_i + 1)}$

$$shift(M, V) = \begin{bmatrix} V_1 & M_{10} & \cdots & M_{1(\sigma_i - 1)} \\ V_2 & M_{20} & \cdots & M_{2(\sigma_i - 1)} \\ \vdots & \vdots & \ddots & \vdots \\ V_m & M_{m0} & \cdots & M_{m(\sigma_i - 1)} \end{bmatrix}.$$

Lastly, $\Lambda_i : S_i \rightarrow Y_i$ is the *output function*, with $\Lambda_i(p_{i_k}) = \begin{cases} 1 & if\ p_{i_{k-1}} \geq \tau_i \\ 0 & otherwise. \end{cases}$

This formalization allows for the characterization of each neuron through a parameter triplet $(\tau_i, r_i, \sigma_i) \in \mathbb{Q} \times \mathbb{Q} \times \mathbb{N}_0$, i.e., the firing threshold τ_i, the remaining coefficient r_i, and the integration window σ_i. Notice that, contrarily to Definition 2, a synchronous reactive neuron is no more directly time-dependent, according to the discrete time representation in synchronous models.

3 A quick introduction to Lustre

A Lustre program is a system of equations defining variables, which are functions from time to their domain of values. Since Lustre respects the synchrony hypothesis, time can be projected onto the set of naturals, making variables infinite sequence of values. Then a program may be viewed as a net of operators and this makes Lustre a data flow language, in which operators react instantaneously to their inputs. Lustre is a functional language operating on *flows*, which are pairs of possibly infinite sequences of values of a given type and *clocks* representing sequences of instants. A program has a cyclic behavior and, at its nth execution cycle, all the involved flows take their nth values.

The unit in Lustre language is called a *node*. Lustre nodes compute output variable sequences of values from input variable ones with equations, expressions and assertions. Variables are typed

and those which do not correspond to inputs must only have one definition in the form of an equation. The equation ‘‘`X = E`’’ defines the variable `X` as being identical to the expression `E`. Both `X` and `E` have the same sequence of values and clock [2]. Lustre language has a few elementary basic types (Boolean, integer, real) and complex external types can be declared. Usual operators over basic types are available: `+, -, ...`; `and, or, not`; `if then else`. These are called data operators and only operate on operands sharing the same clock. They operate point wise on sequences of values of their operands.
For instance, if $X = x_1, x_2, x_3, \dots$ and $Y = y_1, y_2, y_3, \dots$, then the expression $X + Y$ is the flow: $x_1 + y_1, x_2 + y_2, x_3 + y_3, \dots$.

Moreover, Lustre has operators to deal with the logical time represented by clocks. The two main temporal operators are `pre` and $\rightarrow$:

- `pre` (for previous) acts as a memory : if $(e_1, e_2, ..., e_n, \dots)$ is the flow `E`, `pre(E)` is the flow $(nil, e_1, e_2, ..., e_n, \dots)$, nil being an undefined value denoting uninitialized memory.
- $\rightarrow$ (meaning “followed by”) complements the `pre` operator and allows to avoid uninitialized memory: let `E` $= (e_1, e_2, \dots, e_n, \dots)$ and `F` $= (f_1, f_2, \dots, f_n, \dots)$ be two flows, then `E`$\rightarrow$`F` is the expression $(e_1, f_2, \dots, f_n, \dots)$.

Equations help us to define output variables in nodes. Besides, to force variable values, assertions may complement equations. Assertions consist in Boolean expressions that should be always true and they allow to make assumptions on the environment. For instance, the assertion: `assert (true` $\rightarrow$ `not(x and pre(x)))` says that `x` is not true twice consecutively in its value sequence.

Syntactically, a node falls into two parts.

1. A declarative part, where typed input variables are specified as well as typed output variables with the keyword `returns`:
   `node` COUNTER (init, incr: int; reset : bool) `returns` (n: int)
   This declarative part can also involve local variable declarations with the syntax:
   `var` x, y : bool ;
2. The body part, which is a set of unsorted equations and assertions surrounded by the keywords `let` and `tel`.

Finally, another syntactic consideration concerns arrays. They have been introduced in Lustre as a syntactic facility and they are expanded into several variables at compile time. As a consequence, the array dimensions must be specified statically. For instance, the node:

```
node ADD4 (a0,a1,a2,a3: bool; b0,b1,b2,b3: bool)
          returns (s0,s1,s2,s3:bool; carry: bool);
var c0, c1, c2, c3:bool;
let
  (s0, c0) = ADD1(a0, b0, false);
  (s1, c1) = ADD1(a1, b1, c0);
  (s2, c2) = ADD1(a2, b2, c1);
  (s3, c3) = ADD1(a3, b3, c2);
  carry = c3;
tel
```

where `ADD1` is a single adder:

[2] The equation `X = E` means that $\forall n \; X_n = E_n$.

```
node ADD1 (a,b,c1: bool) returns(s,c0:bool)
let
   s = a xor b xor c1;
   c0 = (a and b) or (b and c1) or (c1 and a);
tel
```

can have a simplified syntax, thanks to array use:

```
node ADD4 (A,B: bool^4) returns (S: bool^4; carry: bool);
var C: bool^4;
let
  (S[0], C[0]) = ADD1(A[0], B[0], false);
  (S[1..3], C[1..3] = ADD1(A[1..3], B[1..3], C[0..2]);
  carry = C[3];
tel
```

In this last definition, `bool^4` denotes the type "array of 4 Booleans", indexed from 0 to 3, and the second equation replaces the three equations of the first version. It is expanded exactly as these three equations at compile time.

4 Encoding Neuronal Archetypes in Lustre

The basic archetypes we take into account are the following ones (see Fig. 1).

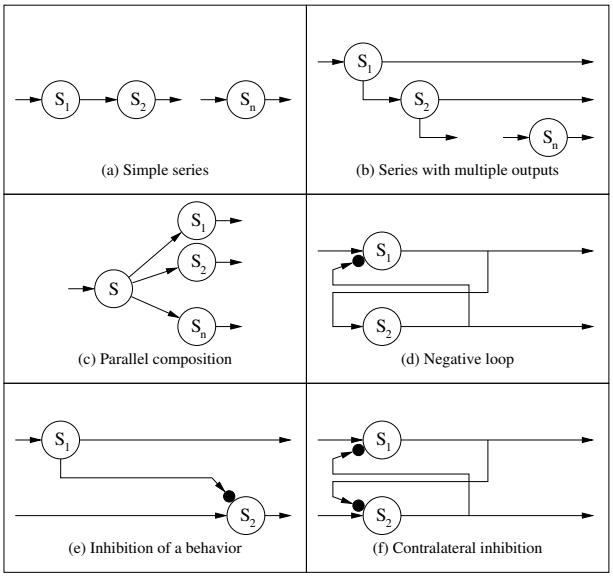


Figure 1: The basic neuronal archetypes.

- **Simple series.** It is a sequence of neurons where each element of the chain receives as input the output of the preceding one. The input (resp. output) of the first (resp. last) neuron is the input (resp. output) of the network. The spike emission of each neuron is constrained by the one of the preceding neuron.

- **Series with multiple outputs.** It is a series where, at each time unit, we are interested in knowing the outputs of all the neurons (i.e., all the neurons are considered as output ones).
- **Parallel composition.** There is a set of neurons receiving as input the output of a given neuron. All neurons working in parallel are considered as output ones.
- **Negative loop.** It is a loop consisting of two neurons: the first neuron activates the second one while the latter inhibits the former one. The inhibited neuron is supposed to oscillate.
- **Inhibition of a behavior.** There are two neurons, the first one inhibiting the second one. After a certain delay, the first neuron is supposed to be activated and the second one to be inhibited.
- **Contralateral inhibition.** There are two or more neurons, each one inhibiting the other ones. The expected behavior is of the kind "winner takes all", that is, starting from a given time only one neuron becomes (and stays) activated and all the other ones are inhibited.

In the following we provide a Lustre implementation of neurons and archetypes. A Boolean neuron with one predecessor (that is, one input neuron), can be modeled as the Lustre node described in Program 1.

Program 1 Basic neuron node.

```
node neuron105 (X:bool) returns(S:bool);
  var
    V:int;
    threshold:int;
    w:int;
    rvector:  int^5;
    mem:int^1*5;
    localS: bool;
  let
    w=10; threshold=105; rvector=[10,5,3,2,1];
    mem[0]=if X then w else 0;
    mem[1..4]=0^4->if pre(S) then 0^4 else pre(mem[0..3]);
    V=mem[0]*rvector[0]+mem[1]*rvector[1]+mem[2]*rvector[2]
       +mem[3]*rvector[3]+mem[4]*rvector[4];
    localS=(V>=threshold);
    S= false -> pre(localS);
  tel
```

In the node `neuron105` (where the firing threshold is set to 105), X is the Boolean flow representing the input signal of the neuron, `w` is the synaptic weight of the input edge, `rvector` is the vector containing the different values the remaining coefficient can take along the integration window (from the biggest to the smallest one), and the vector `mem` keeps trace of the received signals (from the current one to the one received at the time $t - \sigma$)[3]. More precisely, at each time unit the first column of vector `mem` contains the current input (multiplied by the synaptic weight of the input edge) and, for each `i` greater than 0, the value of the column i is defined as follows: (i) it equals 0 at the first time unit (initialization) and (ii) for all following time units it is reset to 0 in case of spike emission at the preceding time unit and it takes the previous time unit value of the column $i - 1$ otherwise. Variable `localS` is used to introduce a delay in the spike emission.

[3]Observe that all the parameters are multiplied by 10 in order to only deal with integer numbers (and thus to be able to use all the model checkers available to Lustre).

The generalization to a node with m predecessors is straightforward. Thanks to the modularity of Lustre, archetypes can be easily encoded starting from basic neurons. As an example, a simple series composed of three (resp. four) neurons of type `neuron105` is described in Program 2 (resp. 3).

Program 2 Simple series of three neurons.

```
node series3 (X:bool) returns(S:bool);
  var
    chain:bool^3;
  let
    chain[0]=neuron105(X);
    chain[1..2]=neuron105(chain[0..1]);
    S=chain[2];
  tel
```

Program 3 Simple series of four neurons.

```
node series4 (X:bool) returns(S:bool);
  var
    chain:bool^3;
  let
    chain[0]=neuron105(X);
    chain[1..3]=neuron105(chain[0..2]);
    S=chain[3];
  tel
```

In the nodes `series3` and `series4`, each position of the vector `chain` refers to a different neuron of the chain. As far as the first neuron is concerned, it is enough to call the node `neuron105` with the input of the series as input. For the other neurons of the chain, their behavior is modeled by calling `neuron105` with the output of the preceding neuron as input. The output of the node is the one of the last neuron of the series.

5 Encoding and Verifying Temporal Properties of Archetypes in Lustre

The behavior of each archetype can be validated thanks to the use of model checkers such as Lesar, Nbac, Luke, Rantanplan, and kind2 (the last four ones have been used to deal with some properties involving integer constraints Lesar is not able to treat). For each archetype, one or two properties have been encoded as Lustre nodes and tested on some instances of the archetype. Most of the properties are tested here for all possible inputs and one or more set(s) of parameters for the given archetype.

5.1 Simple Series (see Fig. 1(a))

Given two series with the same triplets of parameters $(\tau, r, \sigma) \in \mathbb{Q} \times \mathbb{Q} \times \mathbb{N}_0$ and the same synaptic weights, the first series being shorter than the second one, we want to check whether the first series is always in advance with respect to the second one. More precisely, the property we test is the following one:

Property 1. [Comparison of series with same parameters] Given two series with the same neuron parameters and different length (i.e., with a different number of neurons), at each step, the number of spikes emitted by the shorter series is greater or equal than the number of spikes emitted by the longer one.

The node `prop1` (described in Program 4) expresses an observer of Property 1 in Lustre.

Program 4 Observer of Property 1

```
node prop1(X:bool) returns(S:bool);
  var
    A1,A2:bool;
    C1,C2;
  let
    A1=seriesA_sp(X);
    A2=seriesB_sp(X);
    C1=bool2int(A1)->if A1 then pre(C1)+1 else pre(C1);
    C2=bool2int(A2)->if A2 then pre(C2)+1 else pre(C2);
    S=(C1-C2)>=0;
  tel
.
```

Let `seriesA_sp` (resp. `seriesB_sp`) be the Lustre node encoding the first (resp. second) series (corresponding neurons in the two series have the same parameter triplets). In the node `prop1`, `C1` (resp. `C2`) keeps trace of the number of spikes emitted by the first (resp. second) series until the current time unit. The model checkers Lesar, Nbac, Luke, Rantanplan and kind2 verify whether, *whatever is the value of the input flow variable* X (which is common to the two series), the property is true, that is, `C1` is greater or equal than `C2`.

Another interesting property concerning simple series is the following one:

Property 2. [Comparison of series with different parameters] Given two series with different neuron parameters and different length, they always have the same behavior.

The node `prop2` (described in Program 5) encodes such a property in Lustre.

Program 5 Observer of Property 2

```
node prop2 (X:bool) returns(S:bool);
  var
    s1,s2:bool;
  let
    s1=seriesA_dp(X);
    s2=seriesB_dp(X);
    S=(s1=s2);
  tel
```

At each step, the output of the node is true if the output of the two series `seriesA_dp` and `seriesB_dp` is the same (provided that they receive the same input flow X). Such a property can be exploited in order to reduce a given neural network (if a given series has exactly the same behavior than a shorter one, it can be replaced by the second one). As an example, we found a series of 3 neurons showing the same behavior than a series of length 4 (neurons in the two series have different firing thresholds and synaptic weights).

5.2 Series with Multiple Outputs (see Fig. 1(b))

When dealing with a series with multiple outputs, we are interested in checking whether, soon or later, all the neurons of the sequence are able to emit a spike. It may not be the case if the parameters are not well chosen (for example, if the threshold of the first neuron is too high). The corresponding property formalized here is the following one:

Property 3. [Firing ability in a series] Given a series with multiple outputs where the different neurons can have different parameters, there exists a time unit such that all the neurons have emitted.

The node `prop3` (described in Program 6) encodes Property 3.

Program 6 Observer of Property 3

```
node prop3(X:bool) returns(S:bool);
  var    A:bool^4;fb
    A0,A1,A2,A3,F0,F1,F2,F3:bool;
  let
    A=seriem(X);
    A0=A[0];
    A1=A[1];
    A2=A[2];
    A3=A[3];
    F0=after(A0);
    F1=after(A1);
    F2=after(A2);
    F3=after(A3);
    S=F0 and F1 and F2 and F3;
  tel
```

Let `seriem` be a series consisting of 4 neurons and let `after` be a Lustre node whose output variable becomes and remains true the step after an input variable becomes true (that is, the step after a certain event happens). The output of `prop3` becomes (and stays) true after all the neurons of the series have emitted at least one spike. As an example of property violation, we have found a series of length 4 where, even if a flow of 1 (encoded as *true* in Lustre) is given as input, the last neuron is never able to emit. Observe that, given a series where all the neurons are able to emit, `prop3` only becomes true when the last neuron of the series emits a spike. In order to force the property to be immediately true, it is possible to take advantage of the node `always_since` from Lustre distribution library described in Program 7.

Program 7 Node always_since (from Lustre distribution library)

```
node always_since(C,A: bool) returns (X: bool);
  let
    X=if A then C
    else if after(A) then C and pre(X)
    else true;
  tel
```

Such a node requires a first Boolean variable to be always true starting from the instant when a second Boolean variable becomes true. we can introduce a new output variable `SS` defined as `always_since(S,F3)`.

5.3 Parallel Composition (see Fig. 1(c))

We are interested in knowing a lower and an upper bound to the number of neurons that can emit a spike at each time unit. The lower (resp. upper) bound is not necessarily 0 (resp. the number of parallel neurons). More precisely, the property we test is Property 4 (modeled by the Lustre node `prop4` described in Program 8):

Property 4. [Lower/upper firing bounds in a parallel composition]
Given a parallel composition of neurons, all with the same parameters, at each time unit, the number of emitted spike is in between a given interval.

Program 8 Observer of Property 4

```
node prop4 (X:bool; n1,n2:int) returns(S:bool);
  var
    nspike:int;
  let
    nspike=parallel(X);
    S=(nspike<n2) and (nspike>n1);
  tel
```

Let `parallel` be a node encoding a parallel composition of neurons and let the output variable of such a node represent the global number of spikes emitted at each time unit by all the neurons in parallel. The node `prop4` checks whether the number of emitted spikes is always in between a lower bound and an upper bound. As an example, we have found a parallel composition of 3 neurons where the number of emitted spikes is always strictly lower than 3 (more precisely, it is always in between 0 and 2). This is due to the fact that, for one of the parallel neurons, the synaptic weight of the corresponding input edge is too low and it is never able to emit.

5.4 Negative Loop (see Fig. 1(d))

In this case the inhibited neuron is expected to oscillate. In Property 5 we express an oscillation with a period of two time units.

Property 5. [Oscillation in a negative loop] Given a negative loop (where the two neurons do not necessarily have the same parameters), the inhibited neuron oscillates with a pattern of the form *false*, *false*, *true*, *true*.

Such an oscillation is modeled in the node `prop5` (described in Program 9).

Program 9 Observer of Property 5

```
node prop5(X:bool) returns(S:bool);
  var
    S1,S2,Out:bool;
  let
    Out=retroaction(X);
    S1=true->pre(Out);
    S2=true->pre(S1);
    S=if Out then not S2 else S2;
  tel
```

Let `retroaction` be the Lustre node encoding the negative loop archetype and let his output `Out` be the output of the inhibited neuron. In the node `prop5` we check that (i) if `Out` is true, then it was false two time units ago and (ii) if `Out` is false then it was true two time units ago. For several parameters, if we inject only 1 as input of the archetype, such a property is satisfied. Observe that, to test the property satisfaction under some specific conditions, e.g., when the input variable `X` is equal to true, it is sufficient to introduce a new output variable `SS` defined as the disjunction of the current output variable `S` and the negation of the condition. (e.g., `SS=S or X=false`).

5.5 Inhibition of a Behavior (see Fig. 1(e))

To validate this archetype we need to verify that, at a certain instant, the inhibited neuron stops emitting spikes. In particular, the property we encoded is the following one:

Property 6. [Fixed point inhibition] Given an inhibition archetype (where the two neurons do not necessarily have the same parameters), at a certain time the inhibited neuron can only emit *false* values.

The node `prop6` encodes such a property (described in Program 10).

Program 10 Observer of Property 6

```
node prop6(X1,X2:bool) returns (OK:bool);
  var
    S,preS: bool;
    Out:bool^2;
  let
    Out=inhib(X1,X2);
    S=Out[1]; --output of the inhibited neuron
    OK=true-> (not S or S and preS);
    preS=false->pre(S);
  tel
```

Let `inhib` be the node for the inhibition archetype. The output of the node `prop6` is true if the variable representing the output of the inhibited neuron of the archetype cannot pass from `false` to `true`. For appropriate parameter values, if we inject only 1 values, such a property turns out to be true.

5.6 Contralateral Inhibition (see Fig. 1(f))

For such an archetype the expected behavior is of the kind "winner takes all", that is, at a certain point only one neuron is activated and the other ones cannot emit spikes. Such a behavior is described by Property 7 (node `prop7` described in Program 11).

Property 7. [Winner takes all in a contralateral inhibition] Given a contralateral inhibition archetype with two neurons (where the two neurons do not necessarily have the same parameters), at a given time, one neuron is activated and the other one is inhibited.

Program 11 Observer of Property 7

```
node prop7(X1,X2:bool) returns (OK:bool);
  var
    Out:bool ^2;
    N0,N1:bool;
  let
    Out=contralateral(X1,X2);
    N0=Out[0];
    N1=Out[1];
    OK=N0 and not N1 or N1 and not N0;
  tel
```

Let `contralateral` be the Lustre node encoding a contralateral inhibition with two neurons. In the node `prop7` we test whether, at each time unit, one neuron is activated and the other one inhibited. Such a property turns out to be true for several parameters (if only 1 values are injected). Let $w2$ (resp. $w4$) be the synaptic weight of the inhibiting input edge of the first (resp. second) node. In Fig. 2, blue points represent the pairs $(w2, w4)$ for which the property is verified starting from a time unit lower than or equal to 4 and red points are associated to the pairs for which the property is not verified within 10 time units (for some fixed parameter triplets).

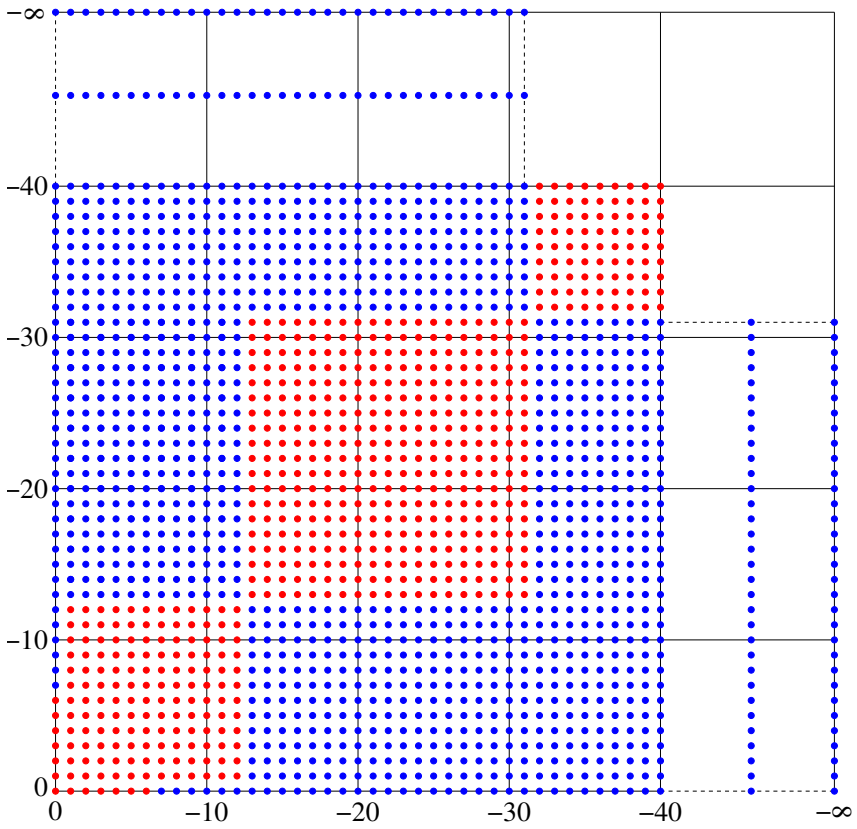


Figure 2: Verification of prop7 for the different values of $(w2, w4)$.

5.7 Comparison of the Model Checkers

A synthesis of the outputs of the five model checkers is summarized for each property in Table 1:

	lesar	nbac	luke	rantanplan	kind2
Simple series (prop1)	No	Yes	very long time!	Yes	Yes
Simple series (prop2)	No	exit before!	Yes	very long time!	Yes
Series with multiple outputs	No	exit before!	Yes	Yes	Yes
Parallel composition	No	exit before!	Yes	Yes	Yes
Negative loop	No	exit before!	Yes	Yes	Yes
Inhibition of a behavior	No	Yes	Yes	Yes	Yes
Contralateral inhibition	No	Yes	Yes	Yes	Yes

Table 1: Comparison of the five model checkers

Notice that, when a model checker gives a negative answer, it does not necessarily mean that the property is false; it can be an indication of the fact that the model checker is not able to conclude. In this experiment, Lesar has difficulties to handle complex integer constraints. Nbac goes further but it is quickly stopped by the polyhedra approach. Luke and its extension Rantanplan give similar results with sometimes a very long computation time. kind2 works quickly and it is able to prove more general properties than Luke and Rantanplan. For instance, Luke and Rantanplan allow for the identification of the pair of weights which stabilize the "Contralateral inhibition" (see Fig. 2) while kind2 is able to straightly give us an infinite set of pair solutions. For the sake of completeness, we also tested the nuXmv model checker [2] but perhaps we could not find the good abstraction (neither too coarse, nor too thorough), so we could not get satisfying results.

6 Discussion and Future Work

In this work, we show how the synchronous language Lustre can be an effective tool to model, specify, and verify neuronal networks. More precisely, we illustrate how some basic neuronal archetypes and their expected properties can be encoded as Lustre nodes and verified thanks to the use of model checkers. For each archetype, we propose one or two representative properties that have been identified after deep discussions with neurophysiologists and, in particular, with the last author of this paper. As a first future work, we intend to propose a more general version of some properties (e.g., expressing oscillation without exactly knowing its period).

We choose to use Lustre because its declarative syntax is more adapted to our class of problems than an imperative language such as Esterel and because several model checkers integrating the symbolic manipulation of integer constraints are at Lustre user's disposition. However, these motivations do not prevent us from considering to use Light-Esterel in the future; the third and fourth author of this work are actually working on extending the expressivity of the declarative part of this language and developing a dedicated model checker. Particularly, this new model checker should integrate a new way to characterize and verify properties based on Linear Decision Diagram (LDD). This representation would allow to identify input parameter intervals of values for which a property holds.

As far as we know, this work constitutes the first attempt to automatically verify the temporal properties of fundamental neuronal archetypes in terms of neuronal information processing (e.g. a negative loop with certain parameters presents a certain oscillating behavior). From there, we will now be able to apply this new approach to all the possible archetypes of 2, 3 or more neurons, up to falling on archetypes of archetypes. One of the questions to ask then will be: are the properties of these archetypes of archetypes simply an addition of the individual constituting archetypes properties or something more? Another one will be: can we understand the computational properties of large

ensembles of neurons simply as the coupling of the properties of individual archetypes, as it is for the alphabet and words, or is there something more again?

Acknowledgements

The authors would like to thank Gérard Berry for an inspiring talk at the *Collège de France* (concerning the checking of temporal properties of neuronal structures) as well as for having indicated us the researchers competent at the use of synchronous programming language libraries (in Sophia Antipolis).

References

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

[2] Nuxmv webpage. `https://nuxmv.fbk.eu/`.

[3] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[4] S. Das. Elements of artificial neural networks [book reviews]. *IEEE Transactions on Neural Networks*, 9(1):234–235, 1998.

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

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

[7] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–9, 2008.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

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

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

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

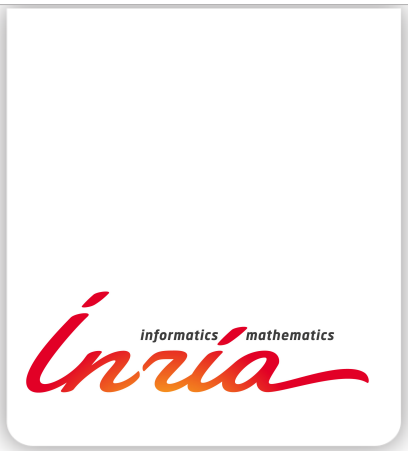
[12] Wolfgang Maass and Technische Universitaet Graz. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8:1–40, 1995.

[13] Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral Approximation of Multivariate Polynomials using Handelman's Theorem. In *International Conference on Verification, Model Checking, and Abstract Interpretation 2016*, St. Petersburg, United States, January 2016. Barbara Jobstmann and Rustan Leino.

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

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

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



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399