View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Queensland eSpace

Modeling Neural Networks using Process Algebras

Robert J. Colvin, January 17, 2013 (10:47 A.M.)
The University of Queensland,
The Queensland Brain Institute,
The Science of Learning Centre,
Brisbane, Australia, 4072

Abstract

Research involving artificial neural networks has tended to be driven towards efficient computation (e.g., in the domain of pattern recognition) or towards elucidating biological processes in the brain (by fitting simulations to experimental data). As our understanding of the biology of individual neurons and the brain as a whole has increased, so have neural network models become more complex, incorporating real-time behaviour, hybrid behaviour (discrete events alongside continuously changing variables), on top of complex system structure and dynamics. To date there has been relatively little effort in fully formally describing neural networks, with typical descriptions in the literature being a mixture of mathematical equations and natural language. This often hides or obscures important aspects of a particular model, and leaves a large conceptual gap between the model descriptions and the usually low-level programming code used to simulate them. In this paper we describe a formal notation and its behaviour which is suited for modelling neural networks. The language is a process algebra, which are well-established mathematical languages for describing highly concurrent (computer) systems and inter-process communication. The basic notions from process algebras are extended with local state variables for maintaining information about individual neurons and synapses, allowing both instantaneous and continuous changes to their values. The notation is used initially to formalise feedforward, backpropagation, and recurrent network behaviour, and then to formalise a more recent neural network model that includes real-time behaviour, differential equations, and complex spatial and temporal relationships between neurons.

1. Introduction

Artificial neural networks have been used for both for their computational power and as abstractions of the behaviour of the brain for many decades. Early forms of neural networks were based on a basic computational principle of a highly connected network of individual neurons that transform their input into an output and send this on to other neurons in the network [1, 2]. This model has attractive computational power, but despite emerging from brain research provides relatively little insight into the biology of real neural networks. More recent research has been on describing the dynamics of the brain and typically involve neurons acting in real-time and governed by differential equations [3, 4], with such models having significant computational power themselves [5, 6]. Although the complexity of the neuron models and network dynamics is quite high, most descriptions of neural networks in the literature tend to mix mathematical equations with natural language, and hence may contain some ambiguity or imprecision. This may be rectified in associated simulation code, but such code is often harder to understand if written in a low-level implementation language, and in particular if it has been optimised for computational efficiency, including the need to recast continuous-time systems as discrete-time systems.

Despite the high degree of communication and concurrency between neurons and the complexity of the behaviour described, there has been relatively little work on fully formalising the description of neural networks. For feedforward networks, perfectly adequate mathematical equations can describe entire systems in one line, however, learning algorithms, the behaviour of recurrent networks, and the dynamics of biologically plausible models are much less concisely described. Models of the latter category, which may include genetic, chemical, and electrophysiological factors, will become more complex and difficult to describe accurately as our knowledge of the brain increases.

The contribution of this paper is in fully formalising some aspects of classic neural networks, as well as a more recent and widely-used neural network model with a closer correlation to neurobiology. To do this effectively we adapt notions of interprocess communication from process algebras ([7, 8, 9]) to handle communication between neurons, extended with local variable declarations to manage changes in, for instance, synaptic strength, and to allow continuous changes to variables in real time. The semantics of the language is described by an operational semantics that gives the behaviour of a process as a list of atomic actions (its *trace*), involving communication with other processes and changes to local variables, as well changes that occur over time. The style of the semantics is also a contribution of the paper, being more compact than related semantics and being amenable to describing the behaviour of a system through the behaviour of its subprocesses.

Sect. 2 gives an introduction to the classic feedforward neural network and backpropagation algorithm in terms of standard mathematical equations. In Sect. 3 we introduce a basic process algebra and use it to recast the mathematical neural network models of Sect. 2, including recurrent neural network behaviour. Sect. 4 formally describes a model of a neural network that includes real-time behaviour, differential equations to describe changes in state over time, and complex temporal relationships between neurons. For these purposes we use a model described by Izhikevich [10], due to its relatively simple (and computationally relatively efficient) local dynamics, but more interesting temporal dynamics and structure (more complex equations, as in the Hodgkin-Huxley model [11], can be substituted without affecting the underlying language or its semantics). In Sect. 5 we give the formal definition of the untimed aspects of the general notation, which is extended in Sect. 6 to a hybrid process algebra.

The motivation of the work is to open the rapidly developing field of neural network modelling to formal analysis by methods from theoretical computer science, by describing neural networks in a uniform and fully formal way. It would be unrealistic to assume that neural network researchers will find immediate benefit in this work, as understanding process algebras and their semantics is a non-trivial undertaking itself; however, as argued by Taylor and Henzinger [12], operational descriptions of biological systems provide the foundation for collaboration between modellers and biologists. In particular, formal modelling may assist in comparing and integrating models drawn from diverse aspects of neural activity.

1.1. Related work

Formally specifying and analysing biological systems is an ongoing research area (e.g., [13, 14, 15, 12, 16, 17, 18, 19]), including recent formal modeling of aspects of cognition [20, 21]. Some of the earliest work on formally specifying classic neural networks is by Smith [22], although a semantics for his framework is not provided. Further work on general formalisations includes predicate-based descriptions in Z [23, 24], process-algebraic descriptions [25], notations with a clearer link to programming languages [26], and those based on automata-like systems [27]. More recently Zaharakis & Kameas [28] developed a general framework for specifying neural networks for use in hardware. The distinction between these papers and our work is that we present a relatively small, abstract language and its semantics, that handles instantaneous events and real-time behaviour, and that we use it to formalise both classic neural networks as well as a more recent biologically plausible model. Some of the above frameworks are arguably flexible and general enough to handle the models presented here, but this is not specifically demonstrated in those presentations.

Hybrid process algebras and semantics have been developed in more general contexts [29, 30]; the formalisation we present here was developed through familiarity with Hybrid$_\chi$ [31]. In comparison with these process algebras, the operational semantics we present is more compact, due to the use of syntactic labels specifying the atomic steps. In particular, this helps to manage the state space, which for neural networks includes many local variables which must be kept distinct between individual neurons. As a result, it is possible to provide relatively concisely the behaviour of processes, using the behaviour of subprocesses to construct the behaviour of the system. The complete traces generated by process algebras with events, local variables, and continuous changes over time are rarely presented in the literature. This aspect of the semantics helps in validation, and provides a further facility for bridging the interdisciplinary divide between biology and formal methods [12].

2. State-based formalisation of a feedforward neural network with backpropagation

In this section we give a specification of a typical feedforward neural network with backpropagation, serving as an introduction to artificial neural networks and formal languages. The description of feedforward and backpropagation

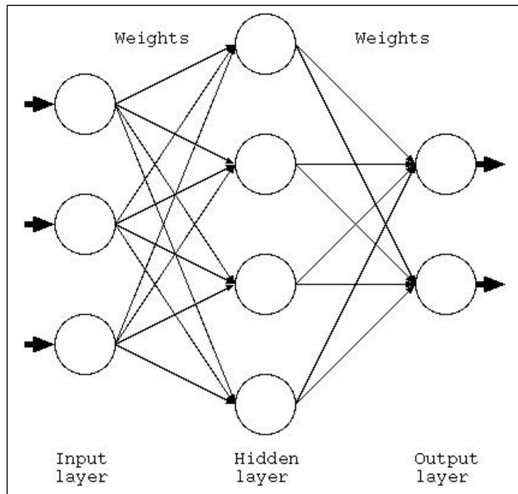

Figure 1: Typical neural network structure

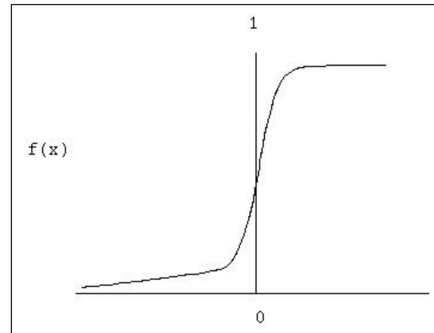

Figure 2: General shape of a sigmoid function

behaviour are derived mainly from Bishop [32].[1]

2.1. Neural network structure

A typical neural network is a function taking a list of 0s and 1s and transforming it into a list of real number values. The function it calculates is an approximation of some otherwise difficult to describe or highly computationally expensive function, for instance character recognition from a pixelated image. The strength of neural networks is that they may be refined through a learning process, without significant intellectual effort on behalf of the "programmer". The disadvantage is that the final function is only an approximation, and the learning process requires a set of examples (input/expected output pairs) of sufficient size.

A typical neural network operates by transforming the input through a series of layers (an input layer, a sequence of hidden layers, and a final output layer), each of which is formed from an array of neurons. Each neuron in layer i is connected to each neuron on layer $i + 1$, and these connections have an associated weight. The notable aspect is that these weights can be automatically fine-tuned during the training process according to particular learning algorithms.

For the purposes of this paper we will take one of the most common structures of a neural network, a multilayer perceptron with a single hidden layer, and hence containing two sets of weights (connecting the input layer neurons to the hidden layer neurons, and connecting the hidden layer neurons to the output layer neurons). A network with three input, four hidden, and two output layer neurons is given in Fig. 1. (For brevity, we now refer to input layer neurons as input neurons, and similarly for hidden layer and output layer neurons.) We also make the following (typical) assumptions about aspects of the network. These assumptions are guided by a principle of keeping the structure of the network reasonably complex but with relatively simple mathematical equations; once the algorithmic structure is specified, substituting more complex (efficient) equations is straightforward.

- Output neurons do not transform their input (i.e., they have a linear activation function).
- Hidden neurons transform their inputs using a sigmoidal activation function (f), the general form of which is shown in Fig. 2, making its derivative ($\dot{f}$) easy to calculate ($\dot{f}(x) = f(x)(1 - f(x))$).
- Errors are calculated using the sum-of-squares approach.

2.2. Feedforward behaviour

The formal mathematical description of a network under the above assumptions (for any number of neurons) is given in Fig. 4 (naming conventions are outlined in Fig. 3). Each neuron is uniquely identified by an element of the

[1] To avoid distraction we do not include bias units in the networks, although their addition is straightforward.

$\mathcal{N}$	The set of neuron identifiers
$\mathcal{I}, \mathcal{H}, \mathcal{O}$	Input, hidden, and output neuron identifiers, mutually exclusive and collectively forming $\mathcal{N}$.
i, h, o	A generic input, hidden, or output neuron identifier
x	A general variable; also specifically an input list with one entry per input neuron
y	A general variable; also specifically the output list with one entry per output neuron
z	The activation (output) of a neuron
$x_i, y_i, z_i, \ldots$	Where x is a function (or list), let x_i abbreviate $x(i)$
w_{hi}	The weight from neuron i to neuron h.
δ_h	The approximate error at neuron h.
f	An activation function used by hidden neurons.
$\dot{f}$	The derivative of function f.
μ	The learning rate (low values induce slower but more stable learning).
κ	A value of any type

Figure 3: Naming conventions

set $\mathcal{N}$, which is partitioned into input, hidden and output neurons ($\mathcal{I}, \mathcal{H}$, and $\mathcal{O}$, respectively). Input x and output y are functions on input neurons and output neurons, respectively, with the input being a list of binary values indexed by the input neurons and the output being a list of real numbers indexed by the output neurons. We use the notation $A \to B$ to denote the type of a function with domain A and range B, hence $x \in \mathcal{I} \to \{0, 1\}$ states that x is a function mapping input neurons to a binary value. If x is a function we let x_i abbreviate $x(i)$. The two layers of real-valued weights are given by w, connecting input to hidden and hidden to output layer neurons. The type $\mathcal{H} \times \mathcal{I}$ is the set of pairs where the first element is in $\mathcal{H}$ and the second is in $\mathcal{I}$. As above, we let w_{hi} abbreviate $w(h, i)$.

The input layer is a set of neurons, $\mathcal{I}$, that encodes the input, x as a set of binary signals. There is little distinction between the input neurons and the input list x, since there is one entry in x for each neuron. The activation of input neuron i is exactly x_i (we say "activation" rather than "output" to avoid confusion with the output layer of neurons). As such there are no specific equations for the input layer. The activation z_h of a hidden neuron h is the linear sum of all the inputs to h (x_i), multiplied by the corresponding weights w_{hi}, and transformed by activation function f (1). The activation of output neuron o, z_o, is calculated similarly (2). The output of the network, the function y, is the list of activations of the output neurons. This can be described with reference to the intermediate activations (3), or the whole network can be summarised in a single equation (4).

Note that we use y' rather than y in (3) and (4), where y' refers to the new value of y after the operation, and y refers to its value before the operation (which in this case is irrelevant). This follows a common specification style that defines a relation between the pre-state and the post-state [33, 34, 35]. For example, a specification that squares x may be written $x' = x^2$, or adding y to x may be written $x' = x + y$. The distinction between pre- and post-states is more important for backpropagation below.

Given a finely tuned set of weights, a function of the form in (4) can efficiently determine whether, for instance, it is likely that a given black and white $n \times n$ image contains a pattern of black pixels that corresponds to some alphanumeric character. In such a case, the input x is a list of binary values representing the (black or white) pixels, and the output is a singleton list containing a real value κ, which may be interpreted as "yes" if $\kappa > 0$ and "no" otherwise. We now present a training method by which the weights can be incrementally updated to give such a function.

2.3. Backpropagation

The behaviour of the network is determined by its weights, and indeed it is the strength of neural networks that a network with a random distribution of weights can be trained to recognise and classify inputs. One such training process is *backpropagation* [36], which we describe below.

During training, in addition to the input x, the expected output t must be provided. Then t is compared to y to determine the size of the $error$. The error is used to calculate new values for the weights, starting from the output layer and "propagating backwards" through the network to the input-hidden weights. For this example we use the well-known sum-of-squares error function, although other choices do not significantly change the structure in Fig. 4.

Assume

$\mathcal{N} = \mathcal{I} \cup \mathcal{H} \cup \mathcal{O} \qquad \mathcal{I}, \mathcal{H}, \mathcal{O}$ are mutually exclusive
$x \in \mathcal{I} \rightarrow \{0, 1\} \qquad y \in \mathcal{O} \rightarrow \mathbb{R}$
$z \in (\mathcal{H} \rightarrow 0..1) \cup (\mathcal{O} \rightarrow \mathbb{R})$
$w \in (\mathcal{H} \times \mathcal{I} \rightarrow \mathbb{R}) \cup (\mathcal{O} \times \mathcal{H} \rightarrow \mathbb{R}) \qquad (w_{hi}, w_{oh} \in \mathbb{R})$
$f \in \mathbb{R} \rightarrow 0..1$ (integrable and differentiable, where $\dot{f}(x) = f(x).(1 - f(x))$)
$t \in \mathcal{O} \rightarrow \mathbb{R} \qquad \delta \in (\mathcal{O} \cup \mathcal{H}) \rightarrow \mathbb{R}$

Feedforward:

$$\forall\, h \in \mathcal{H} \bullet z_h = f\left(\sum_{i \in \mathcal{I}} w_{hi}.x_i\right) \tag{1}$$

$$\forall\, o \in \mathcal{O} \bullet z_o = \sum_{h \in \mathcal{H}} w_{oh}.z_h \tag{2}$$

Execution:

$$(\forall\, o \in \mathcal{O} \bullet y'_o = z_o) \wedge (1) \wedge (2) \tag{3}$$

or

$$\forall\, o \in \mathcal{O} \bullet y'_o = \sum_{h \in \mathcal{H}} w_{oh}.f\left(\sum_{i \in \mathcal{I}} w_{hi}.x_i\right) \tag{4}$$

Backpropagation:

$$\forall\, o \in \mathcal{O} \bullet \delta_o = z_o - t_o \tag{5}$$

$$\forall\, h \in \mathcal{H} \bullet \delta_h = z_h.(1 - z_h).\left(\sum_{o \in \mathcal{O}} w_{oh}.\delta_o\right) \tag{6}$$

$$\forall\, o \in \mathcal{O}, h \in \mathcal{H} \bullet w'_{oh} = w_{oh} - \mu\,.\delta_o.z_h \tag{7}$$

$$\forall\, h \in \mathcal{H}, i \in \mathcal{I} \bullet w'_{hi} = w_{hi} - \mu\,.\delta_h.x_i \tag{8}$$

Execution:

$(3) \wedge (5) \wedge (6) \wedge (7) \wedge (8)$

Figure 4: State-based formal specification of feedforward and backpropagation behaviour

Syntax	Informal meaning
p, e, x, m	Any process; any expression; any variable; any event.
$x := e$	Assign value of e to variable x
$\mathsf{m}(e)!$	Send/generate event m with value e
$\mathsf{m}?$	Receive the value sent via event m ('$\mathsf{m}?$' is an expression)
$(\mathbf{vars}\ \sigma \bullet p)$	Variables in the domain of state σ are local to p
$p_1 \ ; p_2$	Execute process p_1 followed by p_2
p^*	Repeatedly execute p
$p_1 \parallel p_2$	Execute p_1 and p_2 in parallel
$\left(\parallel_{i \in S} p_i \right)$	Multiple processes operating in parallel (one for each element of index set S)
$p_1 \sqcap p_2$	Perform either p_1 or p_2, depending on which is triggered first
$p \backslash A$	Hide the events in the set A from the environment
$\mathbf{nil}$	A terminated process

Figure 5: Basic process syntax

Equation (5) calculates the error at each output neuron o (δ_o), which is the difference between its activation, z_o, and its expected output, t_o. This value is used in equation (6) to calculate the error for hidden neuron h, which is the weighted sum of the errors from the output neurons, multiplied by $\dot{f}(a_h)$, where a_h is the linear weighted sum of the inputs to h. Because f is sigmoidal, $\dot{f}(a_h) = f(a_h).(1 - f(a_h)) = z_h.(1 - z_h)$. Intuitively, these equations state that the error in each neuron is a factor of the proportional effect of that neuron on the error at the eventual output.

Once calculated, the errors at each neuron are used to update the weights, so that the new value of the weight between h and o, w'_{oh}, is the old value (w_{oh}), less the actual output of neuron h (z_h, calculated using the feedforward function) scaled by the error at o (δ_o) and the learning rate (μ) (7). Similarly, the input-hidden weights are updated according the error at h and the activation of the input neurons (8).

The execution of backpropagation requires the calculation of the output y as with the feedforward behaviour (3), and the update of the weights using δ as a temporary variable. The training process is then a matter of providing a sequence of input/expected output pairs, until a reasonable level of convergence is detected, that is, the network does a good job of calculating the expected output. The speed at which a good approximation is reached depends on the number of variables in the input, the size of the network (number of layers and neurons), and the learning rate (μ).

The equations in Fig. 4 are relatively straightforward, giving the behaviour of a conceptual network in Fig. 1. From the perspective of implementing the equations in a program this is a convenient description, however the behaviour of an individual neuron is somewhat obscured. This becomes more of an issue in biologically realistic networks, when the behaviour of individual neurons is better known and easier to specify than the whole network, and where properties of individual neurons may differ significantly.

3. Process-based description

We now consider a process model of the neural network in Sect. 2. The notable difference is that the model is mostly described in terms of the behaviour of individual neurons, the structure of the network is more apparent, and communication between neurons is made explicit. For the straightforward behaviour of a feedforward network (4) this is somewhat of an overkill; for backpropagation and recurrent networks it may be argued that the process model is clearer because the equations are simpler at the individual level rather than the network level; but the effort required is worthwhile for complex models where individual neurons have complex temporal and spatial properties (Sect. 4).

3.1. Syntax

We first informally describe the syntax and semantics of the process language, which is based on CSP [7] for communication and extended by local state [37, 38]. The basic syntax we use for processes is given in Fig. 5 (it is later extended to handle real-time and hybrid behaviour). A formal description of the semantics is deferred until Sect. 5.

A process p may take several forms. An assignment $x := e$ evaluates expression e (atomically) and the value is assigned to variable x. We let $x \mathrel{+}= e$ abbreviate $x := x + e$. A send process $\mathsf{m}(e)!$ atomically evaluates e and sends that value to all processes waiting to receive a value in event m. A receive expression, $\mathsf{m}?$, may appear within an expression in an assignment or send process. When the event is sent by some other process with value κ, the receive expression is replaced by κ.

Given a process containing a receive event $\mathsf{m}?$ as a subexpression, event $\mathsf{m}(\kappa)$ must be received before the process can be executed. For example, assigning to variable x a value "fired" by another neuron may be written $x := \mathsf{fire}?$. This process will be blocked until another process sends a value with event fire, say, $\mathsf{fire}(5)!$, at which point the assignment process will become $x := 5$, and subsequently update x to 5.

The process $(\mathbf{vars}\ \sigma \bullet p)$ where σ is a state – a mapping from variables to values – declares the set of variables in the domain of σ $(\mathrm{dom}(\sigma))$ to be local to p. Any updates to or reads of variables in $\mathrm{dom}(\sigma)$ are performed locally and do not affect any other variables. A state σ is a partial function from variable names (the set Var) to values (the set Val); for instance, a state that maps x to 5 and y to 0 is written $\{x \mapsto 5, y \mapsto 0\}$, and the domain of this state is $\{x, y\}$.

Processes may be sequentially composed, $p_1\ ;\ p_2$, so that after p_1 terminates p_2 begins execution. An infinite iteration of p is given by p^*. Processes may operate in parallel, $p_1 \parallel p_2$, where independent steps of p_1 and p_2 are interleaved, and common events are synchronised. The binary parallel operator may be generalised to $\left(\underset{i \in S}{\parallel} p_i\right)$ where S is some set of (neuron) identifiers, with each process (neuron) p_i operating in parallel. A process choice $p_1 \sqcap p_2$ chooses either p_1 or p_2 for execution depending on which takes a step first, often determined by the environment synchronising on an event offered by either p_1 or p_2. The process $p \backslash A$, where A is a set of events, behaves as p except that events in A are hidden from other processes, i.e., it makes those events local to p. The process $\mathbf{nil}$ indicates a terminated process.

3.2. Feedforward behaviour

The $Network$ is composed of the hidden and output layers working in parallel, with both layers composed from all neurons in that layer themselves working together in parallel (9). The neurons have local variables recording the input from all incoming neurons (x), and corresponding weights (w).[2] This means that the weights are local to each neuron and are distributed throughout the system, rather than collected in a "global" weight variable indexed by pre/postsynaptic pairs of neurons as in Sect. 2. The input is initialised $0_{\mathcal{I}}$, which is a function that returns 0 for each input neuron, and the weights are initialised to some random distribution of values given by the constant W.

The behaviour of a hidden neuron is given by $Hidden_h$ (10): it first stores in x the input list received from the environment via the input event, then fires, sending its activation, given by $f(\Sigma\, x \cdot w)$, to the output layer.[3] The expression $x \cdot w$ is the list formed from the pairwise multiplication of elements in x and w, and $\Sigma\, x$ is the sum of all elements in x, i.e., $\sum_{i \in \mathrm{dom}(x)} x_i$. Thus, letting list multiplication '$\cdot$' have a higher operator precedence than summation 'Σ', we have $\Sigma\, x \cdot w = \sum_{i \in \mathcal{I}} x_i.w_i$.

The output neurons behave similarly (11), however the values received from the hidden layer must be collated into the input list x by receiving each event fire_h in parallel, rather than received in a single block as with the original input. When the parallel process has terminated, the list x contains the relevant value received from each neuron. The output neuron then sends the weighted sum of its inputs to the environment using the output event.

The environment is assumed to interact with the network using a process in the form of $Exec$ (12), where x stores the input value and y records the eventual output of the network, received in parallel from each output neuron.

[2] Note that the weights are associated with incoming edges, that is, the hidden-layer neurons "manage" the weights from the input layer, and the output neurons manage the weights from the hidden layer. This makes the structure simpler for describing feedforward behaviour and is required for the biologically-based network in Sect. 4, but slightly complicates backpropagation, discussed later.

[3] A simpler definition is possible (below), which combines the receiving of the input and the output in the one process, however storing the input value separately is required later for defining backpropagation.

$$Hidden_h = \mathsf{fire}_h(f(\Sigma\ \mathsf{input}? \cdot w))!$$

Assume

$$x \in \mathcal{I} \to \{0,1\} \qquad w \in \mathcal{I} \to \mathbb{R} \qquad \text{(hidden neurons)} \qquad x, w \in \mathcal{H} \to \mathbb{R} \quad \text{(output neurons)}$$

$$\mathrm{w} \in (\mathcal{H} \to \mathcal{I} \to \mathbb{R}) \cup (\mathcal{O} \to \mathcal{H} \to \mathbb{R}) \quad \text{(initial distribution of weights)}$$

$$\mathit{Network} \;\widehat{=}\; \big\| \begin{array}{l} \left(\big\|_{h \in \mathcal{H}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{I}}, w \mapsto \mathrm{w}_h\} \bullet \mathit{Hidden}_h{}^* \right) \\ \left(\big\|_{o \in \mathcal{O}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{H}}, w \mapsto \mathrm{w}_o\} \bullet \mathit{Output}_o{}^* \right) \end{array} \tag{9}$$

$$\mathit{Hidden}_h \;\widehat{=}\; x := \mathsf{input}? \,;\, \mathsf{fire}_h(f(\Sigma\, x \cdot w))! \tag{10}$$

$$\mathit{Output}_o \;\widehat{=}\; \left(\big\|_{h \in \mathcal{H}} x_h := \mathsf{fire}_h? \right) ;\, \mathsf{output}_o(\Sigma\, x \cdot w)! \tag{11}$$

Execution:

$$\mathit{Exec} \;\widehat{=}\; \mathsf{input}(x)! \,;\, \left(\big\|_{o \in \mathcal{O}} y_o := \mathsf{output}_o? \right) \tag{12}$$

Notation:

$$0_S = (\lambda\, i : S \bullet 0)$$

$$x \cdot w = (\lambda\, i : S \bullet x_i . w_i) \qquad \text{assuming } \mathrm{dom}(x) = \mathrm{dom}(w) = S$$

$$\Sigma\, y = \sum_{i \in \mathrm{dom}(y)} y_i \qquad \text{hence if } \mathrm{dom}(x) = \mathrm{dom}(w) = S, \text{ then } (\Sigma\, x \cdot w) = \left(\sum_{i \in S} x_i . w_i \right)$$

$$x_i := e \;\widehat{=}\; x := x \oplus \{i \mapsto e\}$$

$$f \oplus g \;\widehat{=}\; (f \oplus g)(i) = \begin{cases} g(i) & \text{if } i \in \mathrm{dom}(g) \\ f(i) & \text{otherwise} \end{cases}$$

Figure 6: Structure and behaviour of a feedforward network

For instance, some (reactive) program may use the network as a parallel process which can be queried at any time.

$$\big\| \begin{array}{l} (\mathbf{vars}\ \{x \mapsto 0_{\mathcal{I}}, y \mapsto 0_{\mathcal{O}}\} \bullet \text{initialise} ; (\text{construct } x \,;\, \mathit{Exec} \,;\, \text{do something with } y)^*) \\ (\mathit{Network} \backslash \{\mathsf{fire}\}) \end{array} \tag{13}$$

Above we let $\{\mathsf{fire}\}$ represent the set formed from all possible $\mathsf{fire}_h(\kappa)$ events, i.e., $\{\mathsf{fire}_h(\kappa) \mid h \in \mathcal{H} \wedge \kappa \in \mathit{Val}\}$. The hiding has the effect of separating the internal event fire from the environment and avoiding interference, making it explicit that the only communication is through the input and output events. The use of repetition in the client process represents a "batch" mode use of the network. The declarations of x and y in the environment/client process are distinct from the declarations of x (and w) in the neurons in the network process. To avoid distraction the network is written assuming that clients of the network follow the pattern in process (13), that is, they do not send new inputs until the previous inputs have been fully processed.

3.3. Behaviour as traces

We have so far informally described the behaviour of the network, but we now consider a more formal description in terms of the sequence of actions – the *trace* – that a process may take. Each action is a syntactic representation of the step taken, for instance: a property of the state that must hold (e.g., $x = 5$); an update of a variable (e.g., $x := 5$); the name of an event and a value that has been synchronised on (e.g., $\mathsf{fire}_h(5)$); or a combination of the above. A step may also be *silent* (τ), which means it neither relies on nor affects the environment. One of the key aspects of the semantics is that reads and updates of local variables become silent steps at the global level.

After taking a step a process p may be transformed in some way. We write $p \xrightarrow{\ell} p'$ if p takes a step represented by action ℓ, which we also call a label, and evolves to process p'. If the step is silent, we omit it from the label, i.e., $p \longrightarrow p'$ abbreviates $p \xrightarrow{\tau} p'$. We write $p \stackrel{\ell s}{\Longrightarrow} p'$ if p takes the sequence of steps in trace ℓs to evolve to p', and by convention omit silent steps of ℓs. When writing traces the individual labels within ℓs are separated by white space.

3.3.1. The hidden layer

We now look at the behaviour of neurons in the hidden layer (10), repeated below, which occurs within the scope of local variables for recording the input and weights.

$$Hidden_h \mathrel{\widehat{=}} x := \mathsf{input}? \; ; \; \mathsf{fire}_h(f(\Sigma\, x \cdot w))!$$

Consider the behaviour of receiving and storing the input list (value κ).

$$x := \mathsf{input}? \xrightarrow{\mathsf{input}(\kappa)} x := \kappa \xrightarrow{x := \kappa} \mathbf{nil} \tag{14}$$

The first step is a synchronisation on event input with which is passed the value κ, and this value replaces the receive event expression in the assignment. The second step is an update of x to the value κ, after which the process has terminated and may take no further action, as indicated by the process **nil**. Note that labels do not include the decorations for receive ('?') and send ('!') events; the difference in the process syntax determines how to generate the unadorned labels in the traces.

The subsequent behaviour of (10) is to send its activation to the output layer, which we consider below in isolation.

$$\mathsf{fire}_h(f(\Sigma\, x \cdot w))! \xrightarrow{\psi = f(\Sigma\, x \cdot w), \mathsf{fire}_h(\psi)} \mathbf{nil}$$

The label contains two parts, one part being a *guard* that establishes that the activation expression $(f(\Sigma\, x \cdot w))$ has the value ψ in context, and the other part being the synchronisation on fire_h with the value ψ. This transition is therefore highly nondeterministic, with one transition for every possible value ψ, however, only the transition which has the correct value for ψ according to the local variable declaration for x and w will be allowed (i.e., the value $\Sigma\, \kappa \cdot \mathrm{W}_h$).

Joining together the above two traces gives the following trace for a single hidden neuron.

$$Hidden_h \xRightarrow{\mathsf{input}(\kappa) \quad x := \kappa \quad \psi = f(\Sigma\, x \cdot w), \mathsf{fire}_h(\psi)} \mathbf{nil} \tag{15}$$

The repeated behaviour $Hidden_h{}^*$ is an infinite sequence of traces of the above form, one for each new input list to the network, $\kappa_1, \kappa_2, \ldots$.

$$Hidden_h{}^* \xRightarrow{\mathsf{input}(\kappa_1) \quad x := \kappa_1 \quad \psi = f(\Sigma\, x \cdot w), \mathsf{fire}_h(\psi)} Hidden_h{}^* \xRightarrow{\mathsf{input}(\kappa_2) \quad x := \kappa_2 \quad \ldots} Hidden_h{}^* \ldots$$

For each input event to occur, every other process that may synchronise on that event must be ready, hence the second iteration may not begin until all other hidden neurons are ready and the environment sends the event.

Now we consider the behaviour of the hidden neuron inside a local state that declares x and w (as in (9)), i.e., $(\mathbf{vars}\ \{x \mapsto _, w \mapsto \omega\} \bullet Hidden_h{}^*)$. The initial value for x is irrelevant and as such is indicated by '_'. The value ω is a list indexed by the input neurons. The behaviour of the process is similar to that in (15).

$$(\mathbf{vars}\ \{x \mapsto _, w \mapsto \omega\} \bullet Hidden_h{}^*) \xRightarrow{\mathsf{input}(\kappa) \quad \mathsf{fire}_h(f(\Sigma\, \kappa \cdot \omega))} (\mathbf{vars}\ \{x \mapsto \kappa, w \mapsto \omega\} \bullet Hidden_h{}^*)$$

Note that references to variables x and w no longer appear in the labels and have been replaced by their values in the local state, and that x has been updated locally to the new input list κ. This local operation does not appear in the trace, but only indirectly through the new value of x in the state. (For presentation purposes we allow (ground) terms like $f(\Sigma\, \kappa \cdot \omega)$ in the trace to stand for the actual value to which it would evaluate.)

When considered as a whole, the behaviour of the hidden layer involves each hidden neuron synchronising with each other hidden neuron on the input, before synchronising with the output layer. This gives the following behaviour.

$$\left(\big\|_{h\in\mathcal{H}} \mathbf{vars}\ \{x \mapsto _, w \mapsto \omega_h\} \bullet \mathit{Hidden}_h{}^*\right) \xRightarrow{\mathsf{input}(\kappa)\quad \left(\big|\big|\big|_{h\in\mathcal{H}}\ \mathsf{fire}_h(f(\Sigma\,\kappa\cdot\omega_h))\right)} \left(\big\|_{h\in\mathcal{H}} \mathbf{vars}\ \{x \mapsto \kappa, w \mapsto \omega_h\} \bullet \mathit{Hidden}_h{}^*\right) \tag{16}$$

This states that first the input list is received (synchronously) by all hidden neurons, and then the activation for each hidden neuron is sent to the output layer, interleaved in some order. The latter is given by the trace $\left(|||_{h\in\mathcal{H}}\ \mathsf{fire}_h(f(\Sigma\,\kappa\cdot\omega_h))\right)$, which stands for some interleaving of all fire_h events, with the given values. More generally, we write the interleaving of traces ℓs_i as $\left(|||_{i\in\mathcal{I}}\ \ell s_i\right)$, where the local order within each trace ℓs_i is preserved but is interleaved with each other trace ℓs_j. The interleaving of process behaviour is a more abstract (nondeterministic) way of describing behaviour that would typically be implemented as a loop.

3.3.2. The output layer

The behaviour of an output neuron (11) and the output layer (9) is very similar to that of a hidden neuron and the hidden layer, respectively, except that an output neuron must accept each of its inputs (fire_h for $h \in \mathcal{H}$) individually, rather than as a single block. We have the simple trace for receiving each individual value that is analogous to the trace in (14).

$$x_h := \mathsf{fire}_h? \xRightarrow{\mathsf{fire}_h(\chi_h)\quad x_h := \chi_h} \mathbf{nil}$$

Here we have assumed that for each hidden neuron h the value fired, $f(\Sigma\,\kappa\cdot\omega_h)$, is equal to χ_h, where $\chi \in \mathcal{H} \to \mathit{Val}$ collects all such values. Each fire event from the hidden layer is received in parallel and hence results in some interleaving of the individual inputs and updates.

$$\left(\big\|_{h\in\mathcal{H}} x_h := \mathsf{fire}_h\right) \xRightarrow{\left(|||_{h\in\mathcal{H}}\ \mathsf{fire}_h(\chi_h)\quad x_h := \chi_h\right)} \mathbf{nil}$$

The result is that the local variable x is updated precisely to χ.

Each output neuron then sends its own activation to the environment. The resulting trace is similar to (16).

$$\left(\big\|_{o\in\mathcal{O}} \mathbf{vars}\ \{x \mapsto _, w \mapsto \omega_o\} \bullet \mathit{Output}_o{}^*\right) \xRightarrow{\left(|||_{h\in\mathcal{H}}\ \mathsf{fire}_h(\chi_h)\right)\quad \left(|||_{o\in\mathcal{O}}\ \mathsf{output}(\Sigma\,\chi\cdot\omega_o)\right)} \left(\big\|_{o\in\mathcal{O}} \mathbf{vars}\ \{x \mapsto \chi, w \mapsto \omega_o\} \bullet \mathit{Output}_o{}^*\right) \tag{17}$$

3.3.3. The network as a whole

The network (9) involves the two layers operating in parallel, and the behaviour can be constructed from the general transitions given in (16) and (17). These behaviours are merged by synchronising the layers on the fire_h events.

$$\mathit{Network} \xRightarrow{\mathsf{input}(\kappa)\quad \left(|||_{h\in\mathcal{H}}\ \mathsf{fire}_h(\chi_h)\right)\quad \left(|||_{o\in\mathcal{O}}\ \mathsf{output}_o(\psi_o)\right)} \mathit{Network}'$$

Process $\mathit{Network}'$ is $\mathit{Network}$ with changed values for all local variables x. The network receives the input, which is transformed by the hidden neurons and passed to the output neurons, which finally send the computed output back to the environment. In the trace we have assumed that $\psi \in \mathcal{O} \to \mathit{Val}$ is the function where $\psi_o = \Sigma\,\chi\cdot\omega_o$, recalling that $\chi_h = f(\Sigma\,\kappa\cdot\omega_h)$. This may be compared to the mathematical description of feedforward behaviour (3) by noting χ_h and ψ_o are the activations of neurons h and o respectively, i.e., z_h and z_o, and κ is the actual value of the input variable x.

Assume

$$\delta \in \mathbb{R} \quad t \in \mathcal{O} \to \mathbb{R} \quad b \in \mathcal{O} \to \mathbb{R}$$

Let z abbreviate the activation of the local neuron, i.e., in $Train_o$ it abbreviates $\Sigma\, x \cdot w$, and in $Train_h$ it abbreviates $f(\Sigma\, x \cdot w)$.

$$Network^{bp} \;\widehat{=}\; \parallel \begin{matrix} \left(\parallel_{h \in \mathcal{H}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{I}}, w \mapsto \mathrm{w}_h, b \mapsto 0_{\mathcal{O}}, \delta \mapsto 0\} \bullet (Hidden_h \sqcap Train_h)^* \right) \\ \left(\parallel_{o \in \mathcal{O}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{H}}, w \mapsto \mathrm{w}_o, \delta \mapsto 0\} \bullet (Output_o \sqcap Train_o)^* \right) \end{matrix} \tag{18}$$

$$Train_o \;\widehat{=}\; \delta := z - \mathsf{train}_o? \;;\; \left(\parallel_{h \in \mathcal{H}} \mathsf{bprop}_{oh}(\delta . w_h)! \;;\; w_h \mathrel{+}= -\,\mu\,.\delta.x_h \right) \tag{19}$$

$$Train_h \;\widehat{=}\; \left(\parallel_{o \in \mathcal{O}} b_o := \mathsf{bprop}_{oh}? \right) \;;\; \delta := z.(1 - z).\,\Sigma\, b \;;\; \left(\parallel_{i \in \mathcal{I}} w_i \mathrel{+}= -\,\mu\,.\delta.x_i \right) \tag{20}$$

Execution:

$$Exec \;\widehat{=}\; \mathsf{input}(x)! \;;\; \left(\parallel_{o \in \mathcal{O}} \mathsf{output}_o? \right) \;;\; \left(\parallel_{o \in \mathcal{O}} \mathsf{train}_o(t_o)! \right) \tag{21}$$

Notation:

$$x \mathrel{+}= e \quad \widehat{=}\; x := x + e$$

Figure 7: Feedforward network with learning via backpropagation

For the purposes of keeping the interface clean and avoiding interference with other potential uses of the fire event name, the fire event is hidden when the network is placed in parallel with a client process (13). This gives the following trace.

$$Network \backslash \{\mathsf{fire}\} \xRightarrow{\mathsf{input}(\kappa) \quad \left(|||_{o \in \mathcal{O}}\ \mathsf{output}_o(\psi_o) \right)} Network' \backslash \{\mathsf{fire}\}$$

This synchronises straightforwardly with the $Exec$ process in (13). The important point is that the accesses and manipulations of input values and weights are kept local and do not complicate the behaviour at the top level. Similarly, by hiding the internal fire events, the behaviour of the network is abstracted to the input and output events. It is possible to further refine the interface so that only a single output event is communicated, which consolidates all the individual outputs, but for brevity we do not do that here.

3.4. Backpropagation

Now consider a modification of the network to allow learning through backpropagation, as given in (7). The structure of the network (18) stays essentially the same, except that the output neurons and the hidden neurons are extended to allow training behaviour in addition to their normal feedforward behaviour. To accommodate this both types of neurons are extended with a new local variable, δ to record the local error at that neuron. In addition, the hidden neurons include a variable b for consolidating the errors backpropagated from the output neurons into a single list. The training behaviour is included as a choice alongside the usual feedforward behaviour, with the environment selecting which is activated.

An output neuron o is trained (19) by finding the difference between its actual output (z, where z abbreviates $\Sigma\, x \cdot w$) to its expected output, train_o, which is received as an event. The error value, δ, is then sent (backwards) to each hidden neuron, modulated by the weight coming from that hidden neuron. To distinguish backpropagation from

feedforward firing we use the event name bprop. The error value is then used to update the weight according to the learning rule (see (7)).

A hidden neuron h is trained (20) by calculating its own error: the sum of the errors from the output neurons ($\Sigma\, b$) is multiplied by the rate of change in activation at h ($\dot{f}(\Sigma\, x \cdot w)$, where $\dot{f}(x) = f(x).(1 - f(x))$, cf. Fig. 4). The error is then used to update the weights coming from the input layer.

The execution of a network during training (21) is similar to that for feedforward, except that instead of updating the output y the expected output t is used. The input is passed to the network as before, and the output is read from the network although not stored in y (it is irrelevant during training). Once all outputs have been received the training begins, by sending the expected values to the output layer via the event train.

The behaviour of the network is similar to that of feedforward, being based on updates to local variables and synchronising events between and within layers. Outside of the local state declaration the train and bprop events are the only non-silent actions engaged in the behaviour of process Train_o. This is shown below, where σ abbreviates the state $\{x \mapsto \chi, w \mapsto \omega, \delta \mapsto _\}$.

$$(\mathbf{vars}\ \sigma \bullet \mathit{Train}_o{}^*) \xRightarrow{\mathsf{train}_o(\kappa_o)\quad \left(|||_{h \in \mathcal{H}}\ \mathsf{bprop}_{oh}(\psi_{oh})\right)} (\mathbf{vars}\ \sigma' \bullet \mathit{Train}_o{}^*)$$

The state σ' is σ with a new value for δ, and with w updated according to the learning rule. Note that after sending the event bprop, which frees the hidden neurons to begin calculating their own error and updating weights, the output neuron continues by updating its own weights, in parallel with the hidden layer behaviour. There is no interference since the updates are to local variables only in both layers.

The hidden layer behaviour can be derived similarly.

$$(\mathbf{vars}\ \sigma \bullet \mathit{Train}_h{}^*) \xRightarrow{\left(|||_{o \in \mathcal{O}}\ \mathsf{bprop}_{oh}(\psi_{oh})\right)} (\mathbf{vars}\ \sigma' \bullet \mathit{Train}_h{}^*)$$

Combining all behaviour in response to the Exec process (21), including the initial feedforward to calculate the output, we have the following trace, where we let $\mathsf{m}_{\mathcal{I}}$ abbreviate $\left(|||_{i \in \mathcal{I}}\ \mathsf{m}_i(\kappa)\right)$, and $\mathsf{m}_{\mathcal{OH}}$ abbreviate any trace formed from $\left(|||_{o \in \mathcal{O}}\ \left(|||_{h \in \mathcal{H}}\ \mathsf{m}_{oh}(\kappa)\right)\right)$.

$$\mathit{Network}^{bp} \xRightarrow{\mathsf{input}\quad \mathsf{fire}_{\mathcal{H}}\quad \mathsf{output}_{\mathcal{O}}\quad \mathsf{train}_{\mathcal{O}}\quad \mathsf{bprop}_{\mathcal{OH}}} \mathit{Network}^{bp\prime}$$

where $\mathit{Network}^{bp\prime}$ includes updated local weights for the hidden and output neurons.

For training, the bprop events may be hidden in addition to the internal fire events, reducing the visible trace to the following.[4]

$$\mathit{Network}^{bp} \backslash \{\mathsf{fire}, \mathsf{bprop}\} \xRightarrow{\mathsf{input}_{\mathcal{I}}\quad \mathsf{output}_{\mathcal{O}}\quad \mathsf{train}_{\mathcal{O}}} \mathit{Network}^{bp\prime} \backslash \{\mathsf{fire}, \mathsf{bprop}\}$$

3.5. Recurrent neural networks

We now consider computation using a recurrent neural network, which we describe below based on Elman [39], although there are many variants on this approach in the literature. The essential difference with classic neural networks is that recurrent neural networks contain cycles, and as such have a form of short-term memory which means that the output for a given input may differ depending on the preceding inputs. In the simplest case, the top-level network structure stays essentially the same, except that each hidden neuron is now connected to every other hidden neuron (including, for simplicity, itself[5]), and each connection has a corresponding weight, as depicted in Fig. 8.

The input to the network remains as a list of binary values, but now all inputs occur in a discrete time series, so that each input occurs before or after every other input. After (or in parallel) a hidden neuron receives input from the environment, it then receives the *previous* activation of each hidden neuron. The activation of the hidden neuron combines the standard input with the input from the other hidden neurons.

[4] As earlier, in the set of hidden events we allow bprop to abbreviate the set of events $\{\mathsf{bprop}_{oh}(\kappa) \mid o \in \mathcal{O} \wedge h \in \mathcal{H} \wedge \kappa \in \mathit{Val}\}$.

[5] This can be removed by instead quantifying over $\mathcal{H} \setminus \{h\}$.

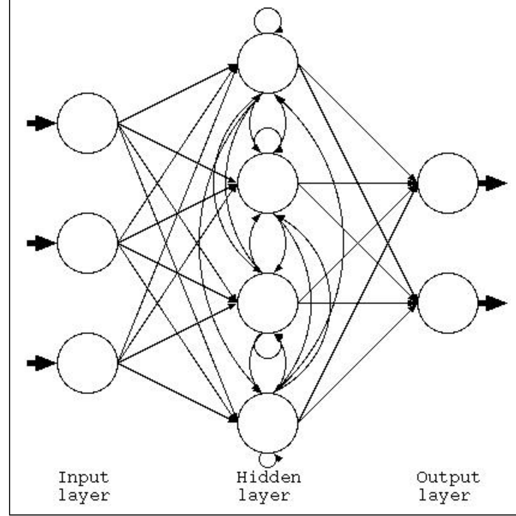

Figure 8: Recurrent neural network structure

Assume

$$x, x^p \in (\mathcal{I} \to \{0,1\}) \cup (\mathcal{H} \to \mathbb{R}) \qquad w \in (\mathcal{I} \cup \mathcal{H}) \to \mathbb{R} \qquad \text{(hidden neurons)}$$

Let z abbreviate $f(\Sigma\, x \cdot w)$ and z^p abbreviate $f(\Sigma\, x^p \cdot w)$.

$$Network^{rc} \;\widehat{=}\; \parallel \begin{array}{l} \left(\parallel_{h \in \mathcal{H}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{I}\cup\mathcal{H}}, w \mapsto \mathrm{w}_h, x^p \mapsto 0_{\mathcal{I}\cup\mathcal{H}}\} \bullet Hidden_h{}^* \right) \\ \left(\parallel_{o \in \mathcal{O}} \mathbf{vars}\ \{x \mapsto 0_{\mathcal{H}}, w \mapsto \mathrm{w}_o\} \bullet Output_o{}^* \right) \end{array} \tag{22}$$

$$Hidden_h \;\widehat{=}\; x_{\mathcal{I}} := \mathsf{input}? \;;\; \left(\mathsf{rfire}_h(z^p)! \parallel \left(\parallel_{j \in \mathcal{H}} x_j := \mathsf{rfire}_j? \right) \right) \;;\; \mathsf{fire}_h(z)! \;;\; x^p := x \tag{23}$$

Notation:

$$x_{\mathcal{I}} := e \;\;\widehat{=}\;\; x := x \oplus e \qquad \text{where } e \in \mathcal{I} \to \mathit{Val} \tag{24}$$

Figure 9: Recurrent network process description

Syntax	Informal meaning
Δt	Delay for t milliseconds.
$\mathsf{m}!, \mathsf{m}?$	Generate or receive event m. Events do not need values in this section.
$(\mathbf{clocks}\ \sigma \bullet p)$	Declares the variables in $\mathrm{dom}(\sigma)$ to be local clocks.
$p^{*\parallel}$	Create new copies of p in parallel
$\mathbf{pr}\ p$	Prioritise events over the passing of time
$\dot{x} = f(x)$	Variable x changes over time according to $f(x)$
$\dot{\vec{x}} = \vec{f}(\vec{x})$	As above, generalised to lists of variables and equations

Figure 10: Hybrid process syntax

The process model is given in Fig. 9. The structure of the network remains the same, with each hidden neuron now recording in x the values received from other hidden neurons as well as the input neurons, and maintains a local variable x^p for remembering the immediately previous inputs to the neuron.

The behaviour of the output neurons remains unchanged from (11). The behaviour of a hidden neuron is extended to accept input from both the environment and other hidden neurons. The input list is stored in $x_{\mathcal{I}}$, that is, all indices in the set $\mathcal{I}$ in x are updated, as a generalisation of the update to a specific index, x_i. The neuron then sends its previous activation, $f(\sum x^p \cdot w)$, to all hidden neurons, using the new rfire_h event, that distinguishes it from the fire event that synchronises with the output layer. In parallel, the hidden neuron receives the rfire events from all hidden neurons, storing the values in x. Once this behaviour is complete, it fires and sends its (current) activation to the output layer, and then records all inputs in x^p for the next run.

The variable manipulations are again all local, and so the internal behaviour of the network is given by the following trace.

$$Network^{rc} \xRightarrow{\mathsf{input}_{\mathcal{I}} \quad \mathsf{rfire}_{\mathcal{H}} \quad \mathsf{fire}_{\mathcal{H}} \quad \mathsf{output}_{\mathcal{O}}} Network^{rc\prime}$$

This has the benefit of showing formally the ordering of events and those that occur in parallel. The execution of the recurrent network remains the same as for feedforward networks (12), with the changes in structure (and additional event name) being hidden from any client.

4. Real-time neural networks

In this section we provide the foundations for describing more biologically realistic neural network models, using elements of real-time and hybrid process algebras. The earlier process descriptions allowed interleaving concurrency, where processes interleave their actions, but not true concurrency, where different processes may be evolving at the same time.

Based on biological experiments, Hodgkin & Huxley [11] developed differential equations that describe the electrical functioning of a neuron membrane, having parameters that correspond directly to chemical and biological aspects of the neuron. These equations have provided the basis for describing the activity of neurons within larger networks, however the large number of variables and complexity of the equations reduce its computational efficiency.

For networks where realistic (real-time) behaviour of neurons is required, but the network structure and its emergent behaviour is of more interest than the behaviour of the neurons themselves, Izhikevich [40, 41] developed a simpler set of differential equations that approximate the electrical activity of a neuron, but using fewer variables and hence being more efficient for simulation. As this paper is less concerned with specific differential equations and is more concerned with developing a general and integrated framework for describing individual neuron behaviour and network behaviour, we will use the Izhikevich equations for individual neurons. At the network level, we will use the *polychronisation* model described by Izhikevich [10] that incorporates spike-timing dependent plasticity (STDP) and a time delay between a neuron spiking and its downstream neurons receiving the spike. However the intention is that other equations can straightforwardly replace the Izhikevich equations or control STDP, and other network behaviour can build on the fundamentals we provide.

$\mathcal{N}$	The set of neuron identifiers
Exc, Inh	Excitatory and inhibitory neuron identifiers, mutually exclusive and collectively forming $\mathcal{N}$.
N_i	Process describing neuron i.
S_{ij}	Process describing the synapse between presynaptic neuron j and postsynaptic neuron i.
v, u	The current voltage of a neuron, and a membrane recovery variable modulating the voltage
I	The synaptic current (input) for a neuron, generated by presynaptic spikes
w, wd	Synaptic weight, and its rate of change
t	A duration (real number), with milliseconds as the base time unit.
t_{pre}, t_{post}	For a given synapse, the time elapsed since the presynaptic (resp. postsynaptic) neuron last fired.
A, B, C, D	Constants controlling the behaviour of the neuron
STDP(t)	Dictates the change in synaptic weight as a function of the time since the last spike (see Fig. 13).

Figure 11: Naming conventions

4.1. Syntax

Before describing a specific model we introduce new process syntax to accommodate the introduction of time, as informally described in Fig. 10 (naming conventions are in Fig. 11). The earlier process syntax is extended by the new process types, and the behaviour of processes is extended so that in addition to taking atomic actions (such as synchronising on an event or updating a variable) they may also delay for some duration (measured in milliseconds). The delay may be guarded, and involve the update of variables at the end of the delay.

The process Δt waits for $t$ms before progressing. This is used for specifying the conductance delay between neurons (related to the physical length of the neuron). The processes m! and m? represent sending and receiving an event. Note that values are not sent with events in this section (and hence receive events are processes rather than expressions), since in the biological model each spike is essentially identical with their relative timing being the important factor. Process (**clocks** $\sigma \bullet p$) is similar to a local variable declaration, except that the variables in the domain of σ are clocks. As with timed automata [42, 43], clocks can be reset, and their value increases linearly with time. Hence the time since an event (spike) can be recorded by resetting a clock when the event occurs.

The process $p^{*\|}$ is similar to a repetition (p^*), except that new instances of p are created in parallel rather than sequentially. Thus, if $p \xrightarrow{\mathsf{m}} p'$, we have

$$p^{*\|} \xrightarrow{\mathsf{m}} (p' \parallel p^{*\|}) \xrightarrow{\mathsf{m}} (p' \parallel p' \parallel p^{*\|}) \ldots$$

This behaviour is used for responding to a sequence of spikes, where the subsequent behaviour of p' takes some non-zero duration.

The process **pr** p is used for prioritising events over the passage of time, and hence is used to ensure that synchronising on events occurs as soon as possible (this property is sometimes called maximal progress or minimal delay; see, e.g., [44, 45]). This process is intended to be used only at the topmost syntactic level.

The most fundamental new hybrid process type is the differential equation, $\dot{x} = f(x)$. This process type never terminates, and instead may delay for a duration t, throughout which the value of x evolves according to the equation $f(x)$. For example, stating that x increases linearly with time may be written as the equation $\dot{x} = 1$. Differential equations control the behaviour of variables as they change over time; however, the value of the variables may also be changed by atomic actions occurring in parallel, for instance, when a spike is received by a neuron its synaptic current gets an immediate increase.

The syntax of differential equations is generalised to describing continuous changes to lists of variables, $\vec{x}$, possibly relying on some other variables $\vec{y}$, written $\dot{\vec{x}} = \vec{f}(\vec{x}, \vec{y})$. In this paper the differential equations we use involve two variables, for instance, u and v, and may depend on one other variable, for instance, I, and we write such generalised equations in the more readable form $\dot{v} = f_1(u, v, I) \wedge \dot{u} = f_2(u, v, I)$. More concretely, the following 2-variable equation, describing the change in voltage of a neuron, is taken from [10] (A_i and B are constants).

$$\dot{v} = 0.04v^2 + 5v + 140 - u + I \quad \wedge \quad \dot{u} = \mathrm{A}_i(\mathrm{B}v - u)$$

Assuming v starts at around -65, and u at around -13, these equations result in v fluctuating slightly around -70 and u around -13. However if the synaptic current I is increased suddenly, e.g., by a spike from an input neuron, the

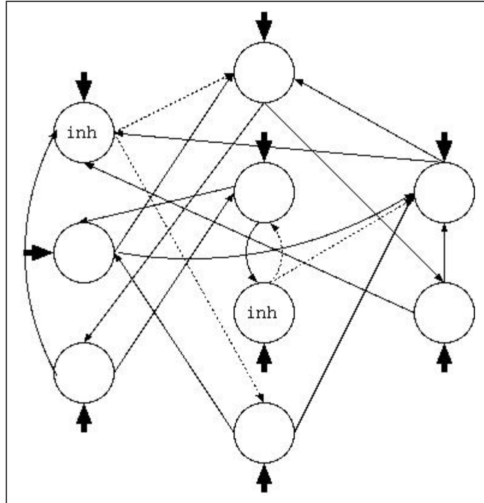

Figure 12: Arbitrary polychronous neural network structure

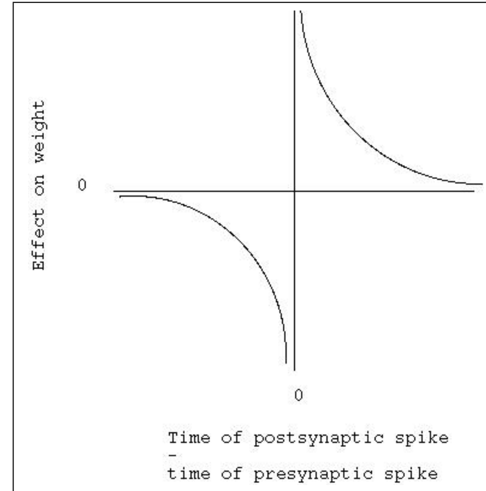

Figure 13: Form of the STDP function

equilibrium may change. If enough input is received to drive v above a threshold, approximately -55 depending on u, then v will continue to increase instead of staying relatively constant.

To simplify the semantics, for a process to be well-formed each variable must be controlled by at most one differential equation.

4.2. The polychronisation model

The neural network described by Izhikevich in [10] is depicted graphically in Fig. 12 and formalised in Fig. 14. The set of neuron identifiers, $\mathcal{N}$, is formed from excitatory (Exc) and inhibitory neurons (Inh), with one fifth of the total neurons being inhibitory. In Fig. 12 there are 10 neurons, 2 of which are inhibitory (marked by inh, with inhibitory connections indicated by dotted lines) and the rest excitatory. For the purposes of the simulations presented in [10] 1000 neurons are used, although this does not otherwise affect the specification. Individual neurons are related by the POST and PRE functions, which define the connections within and hence the structure of the network. For a neuron i, $\text{POST}(i)$ is the set of neurons to which i is connected downstream, i.e., which receive spikes from i. Conversely, $\text{PRE}(i)$ is the set of neurons from which i receives spikes. Neurons are connected by a synapse, S_{ij}, where $i \in \text{POST}(j)$, and we call i the postsynaptic neuron and j the presynaptic neuron. A neuron may not be connected directly to itself (although larger cycles may exist), and an inhibitory neuron may not be directly connected to another inhibitory neuron. For the purposes of simulation in [10], each neuron is connected to 100 neurons downstream (but have more or fewer presynaptic neurons depending on its randomly determined structure).

The connections between neurons represent physical connections of variable length (the length of the axons), and this creates a delay between when the neuron "spikes", and the onset of the associated electrical current in the postsynaptic neurons. The delay from neuron j to neuron i is given by the constant DELAY_{ij} (note that the value is only relevant if $i \in \text{POST}(j)$).

Each synapse has a weight, w, which remains in the range 0..10, although to avoid distraction we do not explicitly state the clipping required to keep w within that range in the model. Each neuron has variable u and v controlling its current voltage, and each synapse has a variable wd which is the rate of change of w. This secondary variable is required as plasticity (learning) changes wd rather than w directly.

The model also makes use of constants $\text{A}, \text{B}, \text{C}$ and D, which control the behaviour of the neuron, and different values of which give different behaviour of the neurons (for instance, how quickly after spiking a neuron may spike again) [40]. In the model in [10], excitatory and inhibitory neurons differ in their values for A and D. The function STDP determines the change in wd as a result of spiking. If $j \in \text{POST}(i)$ and j fires soon after i, that is taken to mean that i was at least partially responsible for causing j to fire, and hence the weight from i to j should be strengthened. On the other hand, if j fires independently of i, then the weight on the synapse should be decreased. This relationship,

Assume

$$\mathcal{N} = Exc \cup Inh \qquad \#Inh/\#\mathcal{N} = 1/5 \qquad (\#N = 1000)$$

$$\textsc{post} \in \mathcal{N} \to \mathbb{P}\mathcal{N} \quad \text{where} \quad i \notin \textsc{post}(i) \qquad i \in Inh \Rightarrow \textsc{post}(i) \cap Inh = \varnothing \qquad (\#\textsc{post}(i) = 100)$$

$$\textsc{pre} \in \mathcal{N} \to \mathbb{P}\mathcal{N} \quad \text{where} \quad \textsc{pre}(i) = \{j \mid i \in \textsc{post}(j)\}$$

$$\textsc{delay} \in (\mathcal{N} \times \mathcal{N}) \to 1..20$$

$$w \in 0..10 \qquad u, v, wd \in \mathbb{R}$$

$$\textsc{a}_i = \begin{cases} 0.02 & \text{if } i \in Exc \\ 0.1 & \text{if } i \in Inh \end{cases} \qquad \textsc{b} = 0.2 \qquad \textsc{c} = \text{-}65 \qquad \textsc{d}_i = \begin{cases} 8 & \text{if } i \in Exc \\ 2 & \text{if } i \in Inh \end{cases}$$

$$\textsc{stdp}(t) = e^{\text{-}t/20}$$

$$Network^b \;\widehat{=}\; \textbf{pr}\left(\mathop{\|}_{i \in \mathcal{N}} N_i\right) \| \left(\Delta 1 \,;\, \mathop{\sqcap}_{i \in \mathcal{N}} \mathsf{input}_i!\right)^* \tag{25}$$

$$\begin{aligned} N_i \;\widehat{=}\; & \textbf{vars}\,\{v \mapsto \textsc{c}, u \mapsto \textsc{b.c}, I \mapsto 0\} \bullet \textbf{clocks}\,\{t_{post} \mapsto 0\} \bullet \\ & \quad \dot{v} = 0.04v^2 + 5v + 140 - u + I \quad \wedge \quad \dot{u} = \textsc{a}_i(\textsc{b}v - u) \\ & \| \; ([v \geq 30] \,;\, \mathsf{spike}_i! \,;\, v := \textsc{c} \,;\, u \mathrel{+}= \textsc{d}_i \,;\, t_{post} := 0)^* \\ & \| \; (\mathsf{input}_i? \,;\, +Current^{20})^* \\ & \| \; \left(\mathop{\|}_{j \in \textsc{pre}(i)} S_{ij}\right) \end{aligned} \tag{26}$$

$[j \in Exc]$

$$\begin{aligned} S_{ij} \;\widehat{=}\; & \textbf{vars}\,\{w \mapsto 6, wd \mapsto 0\} \bullet \textbf{clocks}\,\{t_{pre} \mapsto 0\} \bullet \\ & \quad \dot{w} = wd \quad \wedge \quad \dot{wd} = \text{-}wd/10^4 \\ & \| \; (\mathsf{spike}_i? \,;\, wd \mathrel{+}= 0.1 \times \textsc{stdp}(t_{pre}))^* \\ & \| \; (\mathsf{spike}_j? \,;\, \Delta\textsc{delay}_{ij} \,;\, wd \mathrel{+}= \text{-}0.12 \times \textsc{stdp}(t_{post}) \,;\, t_{pre} := 0 \,;\, +Current^w)^{*\|} \end{aligned} \tag{27}$$

$[j \in Inh]$

$$S_{ij} \;\widehat{=}\; (\mathsf{spike}_j? \,;\, \Delta 1 \,;\, +Current^{\text{-}5})^{*\|} \tag{28}$$

$$+Current^{\kappa} \;\widehat{=}\; I \mathrel{+}= \kappa \,;\, \Delta 1 \,;\, I \mathrel{+}= \text{-}\kappa$$

Figure 14: Polychronous neural network model

described by Song et al. [46], is depicted graphically in Fig. 13. Note that the facilitation and depression values from the STDP function are scaled by 0.1 and 0.12, respectively, biasing the network towards eliminating connections that are not correlated.

The $Network$ is formed from all neurons operating in parallel. In addition to the connections within the network, each neuron i may also receive external inputs (from the thalamus), as signalled by the input_i event. This is the mechanism by which input is given to the network, which would otherwise be quiescent. In [10] the network is driven by input to one randomly selected neuron every millisecond, which we model by a generalised nondeterministic choice between input_i events, $\left(\bigsqcap_{i \in \mathcal{N}} \mathsf{input}_i!\right)$. This process is equivalent to a binary choice between each event, $(\mathsf{input}_1! \sqcap \mathsf{input}_2! \sqcap \ldots)$. More structured input for learning associations between stimuli is explored for a similar network in [47].

A neuron N_i (26) defines local variables v (voltage), u (membrane recovery), and I (synaptic current), and a local clock t_{post} which records the time that has elapsed since i last spiked. The behaviour of u and v over time is given by a complex differential equation, which lets v and u fluctuate at low values. However increases in the synaptic current to i (I, described below) may push v above a threshold of approximately -55, at which point the equations drive v rapidly higher. When v reaches 30, the neuron spikes, the voltage resets and u is increased, and the clock t_{post} is reset. This is given by the repeated process that begins with the guard $[v \geq 30]$. Note that all behaviour of this process takes place instantaneously. The increase in u makes it less likely that v will fire again soon.

Each neuron may receive input from outside the network through the input_i event. If this event occurs, the synaptic current of the neuron jumps by 20 for 1ms, which will typically be enough to cause it to fire, depending on its current values of v and u. This change in synaptic current is given by the process $+Current^{\kappa}$, which increases I by κ, delays for 1ms, and then decreases I again by κ. This process definition can be adapted for more realistic models of synaptic current, such as those of [3, 48].

Each neuron's behaviour is also modulated by the synapse S_{ij} for each of its input neurons $j \in \text{PRE}(i)$. Each synapse where the presynaptic neuron j is excitatory (27) has its own weight w and the current rate of change of w, wd. The synapse also records the time elapsed since the presynaptic neuron (j) spiked. Variable wd is initially 0, and decays slowly over time. It is facilitated (increased) by a spike of the postsynaptic neuron i spikes according to $\text{STDP}(t_{pre})$. If j has recently spiked the increase is relatively large, decaying to 0 the longer it is since j spiked.

When the presynaptic neuron j spikes, there is a delay (corresponding to the time it takes for the spike to travel through the axon), before wd is decreased according to the time elapsed since i spiked, the local clock is reset, and the synaptic current of i is temporarily increased by the strength of the connection from j to i, driving the neuron closer to the spiking threshold (typically it takes at least 2 recent presynaptic spikes to drive a neuron past the threshold). Note the nesting of process S_{ij} within process N_i, which means that S_{ij} may directly affect the variable I declared in N_i.

For a synapse S_{ij} coming from an inhibitory neuron (28), there is no variable weight. Instead, when the inhibitory neuron fires, there is a constant delay of 1ms, before the current of i is decreased by 5 for a period of 1ms. Inhibitory neurons react to their inputs similarly to excitatory neurons (with differences only in the value of the constants in the activity of u) as given in (26).

4.3. Behaviour as traces

To explain the behaviour of the model further we now consider the traces it may generate. In addition to the actions allowed earlier, which in general were given by a label (g, m, η), processes may also take a delay step, given by the label (t, g, η), where t is a duration, g is a guard, and η is a list of updates. For this type of step, guard g must hold initially, then there is a delay of $t$ms duration, at the end of which the update η takes effect. Note that during the delay variables may be changing value, but for the purposes of this paper we need only consider the final value of the variables. As before, we omit g and η from the label if they are the default values ($true$ and ϵ, respectively). A process may therefore either engage in an instantaneous event, or delay for some finite amount of time.

Consider first the behaviour of the simple process $+Current^{\kappa}$. It first updates variable I, then delays for 1ms, and then undoes the effect of the first update to I (spikes have only a temporary effect on the current at postsynaptic targets).

$$+Current^{\kappa} \xRightarrow{I \mathrel{+}= \kappa \quad 1 \quad I \mathrel{+}= -\kappa} \mathbf{nil}$$

The step "1" indicates a delay of exactly 1ms, generated by the $\Delta 1$ process (it is not guarded nor has any associated updates). The delay of 1ms can be formed from a sequence of smaller delays that add up to 1, which may be required if events are interrupting the delay in other parts of the process.

$$\Delta 1 \xrightarrow{0.4 \quad 0.4 \quad 0.2} \mathbf{nil}$$

The superscript κ in process $+Current^{\kappa}$ is a type of by-value parameter, and in (27) we allow a variable in place of κ, $+Current^{w}$. The behaviour is defined by a special rule, which involves evaluating the by-value parameter.[6]

$$+Current^{x} \xrightarrow{x=\kappa} +Current^{\kappa}$$

Now take the behaviour of an inhibitory synapse S_{ij} (a synapse in which the presynaptic neuron j is inhibitory) (28). We define S as follows, and hence $S_{ij} = S^{*\|}$.

$$S \mathrel{\widehat{=}} \mathsf{spike}_j? \ ; \Delta 1 \ ; +Current^{\text{-}5}$$

Process S is blocked until the event spike_j occurs, and it may delay until that time. That is, either of the following transitions is possible, where $t > 0$.

$$S \xrightarrow{t} S \qquad\qquad S \xrightarrow{\mathsf{spike}_j} (\Delta 1 \ ; +Current^{\text{-}5}) \tag{29}$$

Note that this local behaviour potentially allows S to delay indefinitely, which is the required behaviour if neuron j never spikes, however if j does spike we require S to respond immediately. This property can be enforced in a range of ways, but in this paper we enforce it through the $\mathbf{pr}\ p$ command. Because S occurs within a $\mathbf{pr}\ p$ process (25), it will delay until the earliest time at which the spike_j event is permitted by the environment.

After the spike occurs, the process delays for exactly 1ms, and then behaves as $+Current^{\text{-}5}$.

$$(\Delta 1 \ ; +Current^{\text{-}5}) \xrightarrow{1 \quad I \mathrel{+}= \text{-}5 \quad 1 \quad I \mathrel{+}= 5} \mathbf{nil}$$

The behaviour of S_{ij} is S repeated indefinitely in parallel, i.e., $S^{*\|}$. While S delays waiting for spike_j to occur, so too does $S^{*\|}$ delay. However when spike_j occurs a new copy of S (after responding to the event) is created and $S^{*\|}$ remains.

$$S^{*\|} \xrightarrow{t} S^{*\|} \qquad\qquad S^{*\|} \xrightarrow{\mathsf{spike}_j} S^{*\|} \parallel (\Delta 1 \ ; +Current^{\text{-}5})$$

This means that while the execution of $+Current^{\text{-}5}$ is delayed according to the first spike occurrence, further occurrences of spike_j may also be received, creating a queue of delayed $+Current^{\text{-}5}$ processes. For instance, if two spikes are received from neuron j separated by 0.4ms, we get the following trace.

$$\begin{array}{rl}
 & S^{*\|} \\
\xrightarrow{\mathsf{spike}_j} & S^{*\|} \parallel (\Delta 1 \ ; +Current^{\text{-}5}) \\
\xrightarrow{0.4} & S^{*\|} \parallel (\Delta 0.6 \ ; +Current^{\text{-}5}) \\
\xrightarrow{\mathsf{spike}_j} & S^{*\|} \parallel (\Delta 0.6 \ ; +Current^{\text{-}5}) \parallel (\Delta 1 \ ; +Current^{\text{-}5}) \\
\xrightarrow{0.6 \quad I \mathrel{+}= \text{-}5} & S^{*\|} \parallel (\Delta 1 \ ; I \mathrel{+}= 5) \parallel (\Delta 0.4 \ ; +Current^{\text{-}5}) \\
\xrightarrow{0.4 \quad I \mathrel{+}= \text{-}5} & S^{*\|} \parallel (\Delta 0.6 \ ; I \mathrel{+}= 5) \parallel (\Delta 1 \ ; I \mathrel{+}= 5) \\
\xrightarrow{0.6 \quad I \mathrel{+}= 5 \quad 0.4 \quad I \mathrel{+}= 5} & S^{*\|}
\end{array}$$

[6]This may be generalised for any parameterised process straightforwardly following the technique for by-value procedure parameters in [49].

This trace covers a 2.4ms timespan, which, assuming I is initially 0, results in a decrease in I to -5 at 1ms, a further decrease to -10 at 1.4ms, before increasing back to -5 at 2ms and returning to 0 at the end of the 2.4ms.

The behaviour of an excitatory synapse (27) is the parallel composition of three subprocesses. Two subprocesses delay until either a presynaptic or postsynaptic spike occurs, at which time they behave similarly to the inhibitory synapse above. The third subprocess is the following differential equation process W.

$$W \mathrel{\widehat{=}} \dot{w} = wd \quad \wedge \quad \dot{wd} = \text{-}wd/10^4$$

Process W never terminates and simply generates a sequence of delay steps (t, g, η), of varying duration t, where the guard g tests the initial value of w and wd, and the update η is an update of w and wd according the equations over time period t with initial values given in g. For instance, for a delay of 1ms we have the following transition, where $\tilde{n}$ is some number very close to n.

$$W \xrightarrow{1, w=6 \wedge wd=1, (w := \tilde{6} \mid wd := \tilde{1})} W$$

The three subprocesses of S_{ij} must delay in step together, thus, when W delays for exactly 1ms, so too must the processes waiting for spike events as in (29). If we call the other two processes in (27) D_1 and D_2, then if $D_1 \xrightarrow{1} D_1$ and $D_2 \xrightarrow{1} D_2$ we have

$$W \parallel D_1 \parallel D_2 \xrightarrow{1, w=6 \wedge wd=1, (w := \tilde{6}; wd := \tilde{1})} W \parallel D_1 \parallel D_2$$

The guards and updates have been combined, and since for both D_1 and D_2 the guards and updates are trivial the promoted label is the same as that for W.

When this behaviour occurs inside the local clock declaration, the delay has the effect of increasing the value for t_{pre} by 1.

$$(\textbf{clocks}\ \{t_{pre} \mapsto 2\} \bullet W \parallel D_1 \parallel D_2) \xrightarrow{1, w=6 \wedge \ldots} (\textbf{clocks}\ \{t_{pre} \mapsto 3\} \bullet W \parallel D_1 \parallel D_2)$$

When the behaviour is promoted to the local variable declarations for w and wd, the guard and updates are hidden but the duration remains.

$$\begin{array}{ll} & (\textbf{vars}\ \{w \mapsto 6, wd \mapsto 1\} \bullet \textbf{clocks}\ \{t_{pre} \mapsto 2\} \bullet W \parallel D_1 \parallel D_2) \\ \xrightarrow{1} & (\textbf{vars}\ \{w \mapsto \tilde{6}, wd \mapsto \tilde{1}\} \bullet \textbf{clocks}\ \{t_{pre} \mapsto 3\} \bullet W \parallel D_1 \parallel D_2) \end{array}$$

Thus, while no spike events are received for a period of 1ms, synapse S_{ij}'s local weight and wd change slightly, and the time elapsed since the last spike of j is increased. The changes are local and hidden from other processes, although the elapsed time is global and must progress in step with every other synapse and neuron process in the network.

The behaviour of a neuron N_i can be constructed similarly. The differential equation on v and u delays and updates their local values. When v passes 30 a spike event is generated and values of the local variables and clock are reset. The spike event spike_i synchronises with all of N_i's local synapses (coming from all neurons $j \in \textsc{pre}(i)$), as well as synchronising with the relevant synapses in all neurons $j \in \textsc{post}(i)$. Delays are taken in step with each synapse and neuron throughout the network. Neuron N_i will also receive input from outside the network via the event input_i.

For example, the following trace shows the change in v for neuron N_i after presynaptic neurons j and k spike, and as a result drive i itself to spike (the weights on the synapses coming from j and k are at the maximum value, 10). Let N^κ represent process N_i where v has the value κ.

$$N^{\text{-}70} \xrightarrow{1} N^{\text{-}\tilde{70}} \xrightarrow{\mathsf{spike}_j} N^{\text{-}\tilde{60}} \xrightarrow{1} N^{\text{-}\tilde{60}} \xrightarrow{\mathsf{spike}_k} N^{\text{-}\tilde{50}} \xrightarrow{3} N^{\tilde{30}} \xrightarrow{\mathsf{spike}_i} N^{\text{-}65}$$

At the network level the trace of the system are the spikes of each neuron and the duration between each spike. Let $\mathcal{N} = \{j, k, \ldots\}$.

$$\left(\mathop{\parallel}_{i \in \mathcal{N}} N_i \right) \xrightarrow{\mathsf{input}_j \quad 1 \quad \mathsf{spike}_j \quad \mathsf{input}_k \quad 0.2 \quad \mathsf{spike}_l \quad 0.8 \quad \mathsf{input}_m \quad \mathsf{spike}_k \quad \ldots} \ldots$$

From this trace it is straightforward to reconstruct the absolute time of each event, and hence the "spike trains" of each neuron. The input to each neuron (event input_i) is also in the trace. The output of such a network is the pattern of spikes, often localised to a subpopulation of spikes that is taken to correspond to some physical behaviour. For instance, a group of 50 neurons spiking either at the same time or within a short temporal window may correspond to some motor action (flight or fright), in response to the particular temporal pattern of input events. Alternatively, the emergent behaviour of the network under a random sequence of inputs is also of interest; as reported by Izhikevich [10], the network displays oscillations in the total number of spikes firing at any time, corresponding to observed brain activity as measured by electroencephalography (EEG) (for example, [50]).

5. Discrete process semantics

We have so far informally described the meaning of the process language, and shown the behaviour of the processes in terms of traces. We now formalise these notions for the syntax used in Sect. 3 using an operational semantics.

5.1. Syntax

We define the syntax of events, updates, expressions, labels, and processes at the top of Fig. 15. An event $m \in Event$ may be silent/internal, τ, a stand-alone event m, or an event with a value, $\mathsf{m}(\kappa)$, where m is an event name other than τ. This event syntax may appear in the labels on transitions, noting that silent events do not typically appear in the syntax of processes, and events are marked as either input or output in processes using the decorations '!' and '?', respectively.

An expression may be a value, a variable, an input of a value from some other process ($\mathsf{m}?$), or a range of arithmetic and logical expressions such as addition ($e_1 + e_2$, etc.).

An update η is (label-based) syntax for updating 0 or more variables, which in general is written $\vec{x} := \vec{e}$ where $\vec{x}$ is a list of variables and $\vec{e}$ a list of expressions, each of the same length. If the lists are empty we write ϵ. The order of the updates does not matter. We assume that the variables in a list η ($vars(\eta)$) are disjoint, that is, to be well-formed η must not contain more than one update to the same variable. The notation $\eta_1 \mid \eta_2$ represents an interleaving of updates, and is well-formed only if η_1 and η_2 do not share variables.

The general form of a label $\ell \in Label$ is a triple (g, m, η), where the *guard* g is a predicate (boolean-valued expression) that must be satisfied for the transition to take place, m is either silent or an event which much synchronise for the transition to take place, and η is a list of updates to variables. If all three slots are the default values, i.e., $(true, \tau, \epsilon)$, we abbreviate the whole label to τ, and otherwise omit the default slots (see abbreviations in Fig. 15). These labels represent the atomic actions that a process may take. To be well-formed, expressions in labels (in g or in the update expressions) may not contain subexpressions of the form $\mathsf{m}?$. Such subexpressions must be separately evaluated prior to the relevant transition.

The syntax of a process $p \in Proc$ follows that of Fig. 5. A process may be terminated ($\mathbf{nil}$), a guard ($[g]$), an assignment ($x := e$), the sending of (the value of) an expression via an event ($\mathsf{m}(e)!$), a local variable declaration ($\mathbf{vars}\ \sigma \bullet p$), sequential composition ($p_1\ ;\ p_2$) or iteration (p^*), binary ($p_1 \parallel p_2$) or generalised $\left(\underset{i \in S}{\parallel} p\right)$ parallel composition, choice ($p_1 \sqcap p_2$), or event hiding ($p \backslash A$). To be well-formed, processes p_1 and p_2 that synchronise on an event m must not also update the same variable atomically with the synchronisation (as this could cause a conflict), and for a process $p \backslash A$ to be well-formed the set A must not contain τ.

5.2. Semantics

The semantics is given as a set of transition rules in Figs. 15 and 16, in which the labels describe the effect of the (atomic) step taken. The relation

$$\longrightarrow : Label \to (Proc \times Proc)$$

is defined as the smallest relation that satisfies the rules. We write $p \xrightarrow{\ell} p'$ iff $(p, p') \in \longrightarrow (\ell)$.

Rules 1 and 2 are straightforward, with the process syntax being translated directly into label syntax for the step. Both label types interact with enclosing variable declarations as described below. Rule 3 states that a process $\mathsf{m}(e)!$

$Assume \quad \mathsf{m} \in Event \quad A \in \mathbb{P}\, Event \quad x \in Var \quad \kappa \in Val \quad e, g \in Expr \quad \sigma \in Var \nrightarrow Val$

if $x \in T$ then $\vec{x} \in \textbf{list}\ T$

$m ::= \tau \mid \mathsf{m} \mid \mathsf{m}(\kappa)$

$e ::= \kappa \mid x \mid \mathsf{m}? \mid e_1 + e_2 \mid \ldots$

$\eta ::= \epsilon \mid \vec{x} := \vec{e}$

$\ell ::= (g, m, \eta)$

$p ::= \textbf{nil} \mid [g] \mid x := e \mid \mathsf{m}(e)! \mid (\textbf{vars}\ \sigma \bullet p) \mid p_1 \,;\, p_2 \mid p^* \mid p_1 \sqcap p_2 \mid p \backslash A$

Abbreviations: $g \mathrel{\widehat{=}} (g, \tau, \epsilon) \quad \eta \mathrel{\widehat{=}} (true, \tau, \eta) \quad m \mathrel{\widehat{=}} (true, m, \epsilon)$

Rule 1 (Guard).

$$[g] \xrightarrow{g} \textbf{nil}$$

Rule 2 (Assignment).

$$x := e \xrightarrow{x := e} \textbf{nil}$$

Rule 3 (Output event).

$$\mathsf{m}(e)! \xrightarrow{e=\kappa, \mathsf{m}(\kappa)} \textbf{nil}$$

Rule 4 (Input event).

$$p[\![\mathsf{m}?]\!] \xrightarrow{\mathsf{m}(\kappa)} p[\![\kappa]\!]$$

Rule 5 (Local state).

$$\frac{p \xrightarrow{g, m, (\vec{x} := \vec{e} | \eta)} p' \quad \textbf{sat}(g_\sigma) \qquad \vec{x} \subseteq \text{dom}(\sigma) \quad vars(\eta) \cap \text{dom}(\sigma) = \varnothing}{(\textbf{vars}\ \sigma \bullet p) \xrightarrow{(g_\sigma \wedge \vec{e}_\sigma = \vec{\kappa}), m, \eta_\sigma} (\textbf{vars}\ \sigma[\vec{x} := \vec{\kappa}] \bullet p')}$$

Rule 6 (Seq. comp).

$$\frac{p_1 \xrightarrow{\ell} p_1'}{p_1 \,;\, p_2 \xrightarrow{\ell} p_1' \,;\, p_2}$$

Rule 7 (Repetition).

$$p^* \xrightarrow{\tau} p \,;\, p^*$$

Rule 8 (Choice).

$$\frac{p_1 \xrightarrow{\ell} p_1'}{p_1 \sqcap p_2 \xrightarrow{\ell} p_1'} \qquad \frac{p_2 \xrightarrow{\ell} p_2'}{p_1 \sqcap p_2 \xrightarrow{\ell} p_2'}$$

Rule 9 (Hiding).

$$\frac{p \xrightarrow{g, m, \eta} p' \quad m \in A}{p \backslash A \xrightarrow{g, \tau, \eta} p' \backslash A} \qquad \frac{p \xrightarrow{g, m, \eta} p' \quad m \notin A}{p \backslash A \xrightarrow{g, m, \eta} p' \backslash A}$$

Rule 10 (Termination rules).

$$(\textbf{vars}\ \sigma \bullet \textbf{nil}) \xrightarrow{\tau} \textbf{nil} \qquad \textbf{nil} \parallel p \xrightarrow{\tau} \textbf{nil} \qquad \textbf{nil} \sqcap p \xrightarrow{\tau} \textbf{nil} \qquad \textbf{nil} \backslash A \xrightarrow{\tau} \textbf{nil}$$

$$\textbf{nil} \,;\, p \xrightarrow{\tau} p \qquad p \parallel \textbf{nil} \xrightarrow{\tau} \textbf{nil} \qquad p \sqcap \textbf{nil} \xrightarrow{\tau} \textbf{nil}$$

Notation:

$vars(\eta)$		The variables updated in η, $vars(\epsilon) = \varnothing$, $vars(\vec{x} := \vec{e}) = \vec{x}$
$\eta_1 \mid \eta_2$		Some interleaving of the updates in η_1 and η_2, where $vars(\eta_1) \cap vars(\eta_2) = \varnothing$.
$\vec{e} = \vec{\kappa}$	$\Leftrightarrow$	$\vec{e}_1 = \vec{\kappa}_1 \wedge \ldots \wedge \vec{e}_n = \vec{\kappa}_n$ where $\#\vec{e} = \#\vec{\kappa} = n$
$\sigma[\vec{x} := \vec{\kappa}]$		State σ updated so that variables in $\vec{x}$ are mapped corresponding values in $\vec{\kappa}$.
$g[\vec{x} \backslash \vec{e}]$		Replacement of the list of variables $\vec{x}$ by expressions $\vec{e}$ within g
g_σ	$=$	$g[\vec{x} \backslash \sigma(\vec{x})]$ where $\vec{x} = \text{dom}(\sigma)$
$(\vec{x} := \vec{e})_\sigma$	$=$	$\vec{x} := \vec{e}_\sigma \qquad (\epsilon_\sigma = \epsilon)$
$\textbf{sat}(g)$	$\Leftrightarrow$	$\exists\, \sigma \colon Var \rightarrow Val \bullet g_\sigma$

Figure 15: Semantics (discrete system)

sends the current value of e with event m. The value sent, κ, must be equal to e (as enforced by the guard on the label). In all three rules, it is implicit that no input events appear in the relevant expressions of those commands (g, e), else the label will not be well-formed.

Hence, input events must be previously (non-atomically) resolved, as given by Rule 4. We write $p[\![\mathsf{m}?]\!]$ to represent some process p that contains an input event $\mathsf{m}?$ somewhere in its expression. Such a process may synchronise on event $\mathsf{m}(\kappa)$ for any value κ (determined by the corresponding output process through Rule 3), and the event is replaced by κ in p, i.e., $p[\![\kappa]\!]$. For instance, one may specify a guard $[\mathsf{m}? > x]$ that progresses only if the value received via m is greater than some threshold x. It has the following trace by Rules 4 and 1.

$$[\mathsf{m}? > x] \xrightarrow{\mathsf{m}(\kappa)} [\kappa > x] \xrightarrow{\kappa > x} \mathbf{nil}$$

The immediate transition $[\mathsf{m}? > x] \xrightarrow{\mathsf{m}? > x} \mathbf{nil}$ does not use a well-formed label as it contains a decorated event.

Similarly a received value can be used to update a variable, as given by the following trace using Rules 4 and 2.

$$x := f(\mathsf{m}?) \xrightarrow{\mathsf{m}(\kappa)} x := f(\kappa) \xrightarrow{x := f(\kappa)} \mathbf{nil}$$

Rule 5 is the rule for a local state process ($\mathbf{vars}\ \sigma \bullet p$). Consider the general case where p takes a transition $(g, m, (\vec{x} := \vec{e} \mid \eta))$, that is, the guard g must hold, m is the event, and local variables (those in $\mathrm{dom}(\sigma)$) in the list $\vec{x}$ are being updated, with non-local variable updates given in η (recall that the syntax $\eta_1 \mid \eta_2$ specifies some interleaving of the updates in η_1 and η_2). This transition has the effect of updating σ according to $\vec{x} := \vec{\kappa}$, where $\vec{\kappa} \in \mathbf{list}\ \mathit{Val}$ are the evaluations of the expressions $\vec{e}$. However the transition may occur only if $g_\sigma \wedge \vec{e}_\sigma = \vec{\kappa}$ is allowed by the context, where e_σ is e with free variables in the domain of σ replaced by their values in σ. The condition $\mathbf{sat}(g_\sigma)$ requires that there is some possible state in which g_σ may hold, otherwise the transition is not allowed. The updates in the promoted label are the non-local updates in η, again with local variables in the update expressions in η replaced by their local values (η_σ). The event m does not interact with a local state and is promoted as-is.

This rather complex rule may be specialised to cover simpler cases. For instance, consider the case where the transition of p contains a guard only (no event or update).

$$\frac{p \xrightarrow{g} p' \quad \mathbf{sat}(g_\sigma)}{(\mathbf{vars}\ \sigma \bullet p) \xrightarrow{g_\sigma} (\mathbf{vars}\ \sigma \bullet p')} \tag{30}$$

In use with Rule 1 one obtains the transitions $(\mathbf{vars}\ \{x \mapsto 1\} \bullet [5 > x]) \longrightarrow (\mathbf{vars}\ \{x \mapsto 1\} \bullet \mathbf{nil})$. However the process $(\mathbf{vars}\ \{x \mapsto 10\} \bullet [5 > x])$ may not take a step (it is blocked), because $\mathbf{sat}(5 > 10)$ does not hold. In fact this process is blocked indefinitely, as no other process may alter x, but if instead it appears in a larger program, e.g., $(\mathbf{vars}\ \{x \mapsto 10\} \bullet ([5 > x]\ ;\ p_1) \parallel p_2)$, the guard is blocked until process p_2 alters x appropriately.

Now consider the special case of Rule 5 where p updates exactly one variable, with no associated guard or event.

$$(a)\ \frac{p \xrightarrow{x := e} p' \quad x \in \mathrm{dom}(\sigma)}{(\mathbf{vars}\ \sigma \bullet p) \xrightarrow{e_\sigma = \kappa} (\mathbf{vars}\ \sigma[x \mapsto \kappa] \bullet p')} \qquad (b)\ \frac{p \xrightarrow{x := e} p' \quad x \notin \mathrm{dom}(\sigma)}{(\mathbf{vars}\ \sigma \bullet p) \xrightarrow{x := e_\sigma} (\mathbf{vars}\ \sigma \bullet p')}$$

Rule (a) applies when x is local to the state σ, and (b) applies when it is not local. Note that in (a) the promoted label is a guard rather than an update. These rules and Rule 2 give the following transitions.

$$(\mathbf{vars}\ \{x \mapsto 0\} \bullet x := 1) \quad \longrightarrow \quad (\mathbf{vars}\ \{x \mapsto 1\} \bullet \mathbf{nil}) \tag{31}$$

$$(\mathbf{vars}\ \{x \mapsto 0\} \bullet x := y) \quad \xrightarrow{y = \kappa} \quad (\mathbf{vars}\ \{x \mapsto \kappa\} \bullet \mathbf{nil}) \tag{32}$$

$$(\mathbf{vars}\ \{x \mapsto 0\} \bullet y := x) \quad \xrightarrow{y := 0} \quad (\mathbf{vars}\ \{x \mapsto 0\} \bullet \mathbf{nil}) \tag{33}$$

Transition (32) represents many possible transitions, one for each value $\kappa \in \mathit{Val}$. The correct value for y is resolved by the guard using (30) on the outer (not shown) variable declaration containing y. Transition (33) does not update a

$p ::= \ldots \mid p_1 \parallel p_2$ Abbreviation: $\left(\parallel_{i \in 1..n} p_i\right) \mathrel{\widehat=} p_1 \parallel \ldots \parallel p_n$

Rule 11 (Concurrency).

$$(a)\ \frac{p_1 \xrightarrow{g_1, \mathsf{m}(\kappa), \eta_1} p_1' \quad p_2 \xrightarrow{g_2, \mathsf{m}(\kappa), \eta_2} p_2'}{p_1 \parallel p_2 \xrightarrow{g_1 \wedge g_2, \mathsf{m}(\kappa), (\eta_1 | \eta_2)} p_1' \parallel p_2'}$$

$$(b)\ \frac{p_1 \xrightarrow{g, m, \eta} p_1' \quad m \notin \alpha(p_2)}{p_1 \parallel p_2 \xrightarrow{g, m, \eta} p_1' \parallel p_2} \qquad (c)\ \frac{p_2 \xrightarrow{g, m, \eta} p_2' \quad m \notin \alpha(p_1)}{p_1 \parallel p_2 \xrightarrow{g, m, \eta} p_1 \parallel p_2'}$$

Notation:

$\alpha(p)$ The *alphabet* of p, i.e., the set of events upon which p may synchronise.

Figure 16: Semantics (discrete system)

local variable, but the update expression does rely on the local variable x, and its value is reflected in the promoted label $y := 0$ (the term η_σ in the general Rule 5).

Rules 6 and 7 are straightforward for sequential composition of processes, and repetition of processes. Note that repetition is infinite. Rule 8 for $p_1 \sqcap p_2$ executes the first process to take an action, eliminating the other. This is sometimes called *external choice* in the literature, if each process is waiting to synchronise on an event with the environment (i.e., if the label ℓ is an event). Rule 9 for a process $p \backslash A$ follows CSP by hiding any event m in the set A from the environment, which has the effect of making m an internal event of process p. Rule 10 covers the termination rules for the operators, which are based on the interaction of the $\mathbf{nil}$ command with those operators.

Rule 11 for a concurrent process $p_1 \parallel p_2$ states that both processes may transition on actions which contain the same event name $\mathsf{m}(\kappa)$ then the two processes *synchronise* on that event. The promoted label contains the conjunction of the guards (both g_1 and g_2 must be satisfied for the synchronised transition to take place), and the combined updates of each process ($\eta_1 \mid \eta_2$). Recall that to be well-formed, actions that synchronise on the same event must not update the same variables, and hence the set of variables in η_1 and η_2 are disjoint.

Alternatively, rather than synchronising, parallel processes will interleave their actions (Rule 11(b) and (c)). Given a transition $p \xrightarrow{g, m, \eta} p_1'$ where m is not in the *alphabet* of p_2 ($\alpha(p)$), then p_1 may take that transition independently (possibly synchronising on m with some other process). The alphabet of a process p is essentially all events that p may engage in now *or in the future*, and can be extracted syntactically from p [7]. This means that the "sender" of an event is blocked until all receivers are ready (and vice versa).

Note that τ may never appear in the alphabet of a process, and hence a special case of Rule 11(b) is where p_1 is performing an internal action (τ). This means that guard-only actions, assignment-only actions, or actions that combine a guard with an assignment, always interleave.

$$\frac{p_1 \xrightarrow{g, \eta} p_1'}{p_1 \parallel p_2 \xrightarrow{g, \eta} p_1' \parallel p_2} \qquad \frac{p_2 \xrightarrow{g, \eta} p_2'}{p_1 \parallel p_2 \xrightarrow{g, \eta} p_1 \parallel p_2'}$$

The special case of Rule 11(a) where processes synchronise on an event with no (non-trivial) guards or updates is given by the following rule, which covers the majority of cases in this paper.

$$\frac{p_1 \xrightarrow{\mathsf{m}(\kappa)} p_1' \quad p_2 \xrightarrow{\mathsf{m}(\kappa)} p_2'}{p_1 \parallel p_2 \xrightarrow{\mathsf{m}(\kappa)} p_1' \parallel p_2'}$$

Since neural networks are formed from many similar processes working in parallel, we use the notation $\left(\parallel_{i \in S} p_i\right)$ to stand for the parallel composition of n processes, where n is the cardinality of set S. It may be interpreted as the nested binary composition of all the processes p_i, for $i \in S$. Typically all processes p_i have similar behaviours and hence alphabets. We may derive the following rules which show how they interact with each other and with the environment.

$$(a)\ \frac{\forall\, i\colon S \bullet p_i \xrightarrow{\mathsf{m}(\kappa)} p_i'}{\left(\parallel_{i \in S} p_i\right) \xRightarrow{\mathsf{m}(\kappa)} \left(\parallel_{i \in S} p_i'\right)} \qquad (b)\ \frac{\forall\, i \in S \bullet p_i \xrightarrow{\mathsf{m}_i(\kappa_i)} p_i'}{\left(\parallel_{i \in S} p_i\right) \xRightarrow{\left(|||_{i \in S}\ \mathsf{m}_i(\kappa_i)\right)} \left(\parallel_{i \in S} p_i'\right)}$$

$$(c)\ \frac{\forall\, i \in S \bullet p_i \xRightarrow{\left(|||_{j \in T}\ \mathsf{m}_j(\kappa_j)\right)} p_i'}{\left(\parallel_{i \in S} p_i\right) \xRightarrow{\left(|||_{j \in T}\ \mathsf{m}_j(\kappa_j)\right)} \left(\parallel_{i \in S} p_i'\right)}$$

Rule (a) applies when each neuron in a layer can receive the same event, for instance, each hidden neuron receives the event $\mathsf{input}(\kappa)$ from the input layer. Rule (b) applies when each neuron in a layer generates a related event, for instance, each hidden neuron h generates the event $\mathsf{fire}_h(\kappa)$ which is received by the output layer. Rule (c) applies when each neuron in a layer can receive the same set of events, for instance, every output neuron receives every $\mathsf{fire}_h(\kappa)$ event from the hidden layer, for $h \in \mathcal{H}$. In this case the labels are promoted so that the behaviour of the layer is the same as the behaviour of the individual neurons.

6. Hybrid process semantics

We now give the formal semantics of the hybrid process language that combines instantaneous events with variables changing in real-time. The syntax is an extension of the earlier language, and the earlier laws are preserved (i.e., the semantics exhibits semantics conservation [44], or, more generally, modularity [51]). The key difference is that steps may also take a certain duration.

6.1. Syntax

As outlined in Fig. 17, the language is extended by several new process types: Δt for delaying for exactly t time units (milliseconds in this paper) where $t \in \mathbb{R}_{\geq 0}$; parameterless send/receive events, $\mathsf{m}!/\mathsf{m}?$; a local clock declaration $(\mathbf{clocks}\ \sigma \bullet p)$ where each variable in the domain of σ is a clock that increases linearly with the passage of time, and may be set or updated like a normal variable; a process $p^{*\parallel}$ which spawns new copies of p, where typically p is blocked waiting for an event; and $\mathbf{pr}\ p$ which prioritises discrete events over the passage of time. We deal with differential equations separately in Sect. 6.3.

The syntax of labels is extended to include a label type (t, g, η), which represents the passage of t time units, provided g holds at the current time, and after which the state is updated according to η. If the guard is $true$ and the update is empty, the label is abbreviated to t. For such a step to be valid, t must be a non-zero positive real number.

6.2. Semantics

The semantics is given in Fig. 17. Rule 12 states that a process Δt may take any sequence of steps that in total add up to t time units, before terminating. Rule 13 states that an event can be generated straightforwardly by the send/receive syntax (which is used only to convey intent, but in actuality there is no semantic difference). Both action types may also delay indefinitely; however, if the events appear inside a process $\mathbf{pr}\ p$, the event transition will be given priority, as described below.

Rule 14 for process $(\mathbf{clocks}\ \sigma \bullet p)$ states that if p takes a delay step of duration t, the value of the clocks in σ are all advanced by t (notation $\sigma + t$ adds t to all values in σ). Clocks may also be tested and updated in the usual way (see Rule 22 below), but it is expected that clocks will only be assigned real values; local clocks in timed automata [42, 43] are updated only to 0 (reset).

$Assume \quad t \in \mathbb{R}_{\geq 0}$

$p ::= \ldots \mid \Delta t \mid \mathsf{m}! \mid \mathsf{m}? \mid (\mathbf{clocks}\ \sigma \bullet p) \mid p^{*\parallel} \mid \mathbf{pr}\ p$

$\ell ::= \ldots \mid (t, g, \eta) \qquad Abbreviation: \quad t \mathrel{\widehat{=}} (t, true, \epsilon)$

Rule 12 (Delay).

$$\frac{0 < t' \leq t}{\Delta t \xrightarrow{t'} \Delta(t - t')} \qquad \Delta 0 \xrightarrow{\tau} \mathbf{nil}$$

Rule 13 (Events).

$$\mathsf{m}! \xrightarrow{\mathsf{m}} \mathbf{nil} \quad \mathsf{m}? \xrightarrow{\mathsf{m}} \mathbf{nil}$$
$$\mathsf{m}! \xrightarrow{t} \mathsf{m}! \quad \mathsf{m}? \xrightarrow{t} \mathsf{m}?$$

Rule 14 (Clock update).

$$\frac{p \xrightarrow{t,g,\eta} p'}{(\mathbf{clocks}\ \sigma \bullet p) \xrightarrow{t,g,\eta} (\mathbf{clocks}\ \sigma + t \bullet p')}$$

Rule 15 (Parallel repetition).

$$(a)\ \frac{p \xrightarrow{g,m,\eta} p'}{p^{*\parallel} \xrightarrow{g,m,\eta} p' \parallel (p^{*\parallel})} \qquad (b)\ \frac{p \xrightarrow{t,g,\eta} p}{p^{*\parallel} \xrightarrow{t,g,\eta} p^{*\parallel}}$$

Rule 16 (Action precedence).

$$(a)\ \frac{p \xrightarrow{t} p' \quad (\forall p'', t' < t \bullet p \xrightarrow{t'} p'' \Rightarrow p'' \stackrel{m}{\not\longrightarrow})}{\mathbf{pr}\ p \xrightarrow{t} \mathbf{pr}\ p'} \qquad (b)\ \frac{p \xrightarrow{\mathsf{m}} p' \quad p \stackrel{\tau}{\not\longrightarrow}}{\mathbf{pr}\ p \xrightarrow{\mathsf{m}} \mathbf{pr}\ p'} \qquad (c)\ \frac{p \xrightarrow{\tau} p'}{\mathbf{pr}\ p \xrightarrow{\tau} \mathbf{pr}\ p'}$$

Rule 17 (Termination rules).

$$(\mathbf{clocks}\ \sigma \bullet \mathbf{nil}) \xrightarrow{\tau} \mathbf{nil} \qquad \mathbf{pr}\ \mathbf{nil} \xrightarrow{\tau} \mathbf{nil}$$

Rule 18 (Parallel time advance).

$$\frac{p_1 \xrightarrow{t,g_1,\eta_1} p_1' \quad p_2 \xrightarrow{t,g_2,\eta_2} p_2'}{p_1 \parallel p_2 \xrightarrow{t,g_1 \wedge g_2,\eta_1 | \eta_2} p_1' \parallel p_2'}$$

Rule 19 (Choice time advance).

$$\frac{p_1 \xrightarrow{t,g_1,\eta_1} p_1' \quad p_2 \xrightarrow{t,g_2,\eta_2} p_2'}{p_1 \sqcap p_2 \xrightarrow{t,g_1 \wedge g_2,\eta_1 | \eta_2} p_1' \sqcap p_2'}$$

Rule 20 (Hide (delay)).

$$\frac{p \xrightarrow{t,g,\eta} p'}{p \backslash A \xrightarrow{t,g,\eta} p' \backslash A}$$

Rule 21 (Guard delay).

$$[g] \xrightarrow{t} [g]$$

Notation:

$$\sigma + t \quad = \quad (\lambda\, x \colon \mathrm{dom}(\sigma) \bullet \sigma(x) + t) \tag{34}$$

$$p \stackrel{m}{\not\longrightarrow} \quad \widehat{=} \quad \neg(\exists\, p' \bullet p \xrightarrow{m} p') \tag{35}$$

Figure 17: Semantics (hybrid system)

Rule 15 for process $p^{*\|}$ states that a new copy of p is spawned if p may take an action step. Typically p is guarded by an event, and thus whenever this event occurs a new process p appears in parallel with $p^{*\|}$. This process allows multiple copies of p to respond to the environment, possibly executing behaviour taking some period of time to complete. If p can take a delay step the parallel-repetition may also delay.

Rule 16(a) states that the action precedence process $\mathbf{pr}\ p$ allows the passage of time only if no discrete events are possible throughout the duration. This means that inside p time may pass only up to the earliest point at which some discrete action m becomes enabled. Furthermore, Rule 16(b and c) states that silent events are prioritised over named events. For process $\mathbf{pr}\ p$ to be well-formed, process p must not contain reference to undeclared variables, that is, the $\mathbf{pr}\ p$ declaration must occur at the outermost level, and hence each step of p is either a pure event m or a delay t. This constraint simplifies the semantics and is all that is typically required, however a more sensitive version that allows guarded labels may also be defined. The use of action precedence gives the process the *maximal progress* [45] (or minimal delay [44]) property. The rule uses the notation $p \stackrel{m}{\not\longrightarrow}$, which states that p may not take a transition with label m.

Rule 17 states that clock declarations and action precedence operators terminate when their process terminates.

In addition to these new process types, we must add rules to define how the original process types interact with the new delay label. Rule 18 states that parallel processes must advance time at the same rate. For this to happen, both guards must be satisfied by the environment, and both updates take place (recall that to be well formed a variable must not be controlled by more than one differential equation, and hence η_1 and η_2 must operate on disjoint variables). Rule 19 is similar, allowing time to pass in each process in a choice before the selection is made via Rule 8. Rule 20 simply states that hiding events has no effect on a delay step. Rule 21 allows time to pass while a guard is blocked (by Rule 16 it will transition by Rule 1 as soon as the guard evaluates to true).

We must also provide rules for a local clock declaration in an action step, and a local state declaration in a time step. Both rules are straightforward adaptations of Rule 5.[7]

Rule 22 (Local clocks/states).

$$\frac{p \xrightarrow{g,m,(\vec{x} := \vec{e}|\eta)} p' \qquad \mathbf{sat}(g_\sigma) \qquad \vec{x} \subseteq \mathrm{dom}(\sigma) \qquad \mathit{vars}(\eta) \cap \mathrm{dom}(\sigma) = \varnothing}{(\mathbf{clocks}\ \sigma \bullet p) \xrightarrow{g_\sigma \wedge \vec{e}_\sigma = \vec{\kappa}, m, \eta_\sigma} (\mathbf{clocks}\ \sigma[\vec{x} := \vec{\kappa}] \bullet p')} \qquad \frac{p \xrightarrow{t,g,(\vec{x} := \vec{e}|\eta)} p' \qquad \mathbf{sat}(g_\sigma) \qquad \vec{x} \subseteq \mathrm{dom}(\sigma) \qquad \mathit{vars}(\eta) \cap \mathrm{dom}(\sigma) = \varnothing}{(\mathbf{vars}\ \sigma \bullet p) \xrightarrow{t, g_\sigma \wedge \vec{e}_\sigma = \vec{\kappa}, \eta_\sigma} (\mathbf{vars}\ \sigma[\vec{x} := \vec{\kappa}] \bullet p')}$$

6.3. Syntax and semantics of differential equations

Fundamental to the hybrid behaviour of the process algebra is the addition of differential equations as process types, as shown in Fig. 18. In the simplest case, the equation describes the change over time of one variable x, written $\dot{x} = f(x)$.[8] For example, the equation $\dot{x} = 1$ describes a constant increase in the value of x in line with time, i.e., x is effectively a clock, while the equation $\dot{wd} = \text{-}wd/10^4$ describes a very slow trend to 0 for wd, whether positive or negative. The equation process type never terminates, and never engages in an event. The notation can be generalised to a list of variables $\vec{x}$, each of which changes over time as a function $f_i \in \vec{f}$ of $\vec{x}$ and other variables $\vec{y}$, written $\dot{\vec{x}} = \vec{f}(\vec{x}, \vec{y})$.

Rule 23 states that a differential equation process $\dot{x} = f(x)$ may take a delay transition with label $(t, x = \boldsymbol{\kappa}_0, x := \boldsymbol{\kappa}_t)$, where the duration t is arbitrarily chosen, and $\boldsymbol{\kappa}$ is a solution for $\dot{x} = f(x)$ with an initial value for x

---

[7] A more compact alternative that makes it clearer that the semantics of Rule 5 is preserved can be achieved after introducing the atomic combination of a guard and update as a process type, $([g], \eta)$, which is defined by the rule $([g], \eta) \xrightarrow{g,\tau,\eta} \mathbf{nil}$. This can be used to generate the required behaviour (changes to the promoted label and local state) using Rule 5.

$$\frac{p \xrightarrow{g,m,\eta} p' \qquad (\mathbf{vars}\ \sigma \bullet ([g], \eta)) \xrightarrow{g',\tau,\eta'} (\mathbf{vars}\ \sigma' \bullet \mathbf{nil})}{(\mathbf{clocks}\ \sigma \bullet p) \xrightarrow{g',m,\eta'} (\mathbf{clocks}\ \sigma' \bullet p')} \qquad \frac{p \xrightarrow{t,g,\eta} p' \qquad (\mathbf{vars}\ \sigma \bullet ([g], \eta)) \xrightarrow{g',\tau,\eta'} (\mathbf{vars}\ \sigma' \bullet \mathbf{nil})}{(\mathbf{vars}\ \sigma \bullet p) \xrightarrow{t,g',\eta'} (\mathbf{vars}\ \sigma' \bullet p')}$$

[8] We use Newton's notation for the time derivative of x ($\dot{x}$) for its compactness and to avoid notational clashes associated with using a prime.

$$Assume \quad \boldsymbol{\kappa} \in \mathbb{R}_{\geq 0} \to Val \qquad \vec{\boldsymbol{\kappa}} \in \mathbb{R}_{\geq 0} \to \textbf{list}\ Val \qquad (\vec{\boldsymbol{\kappa}}_t \in \textbf{list}\ Val) \qquad \psi \in Val$$

$$p ::= \ldots \mid \dot{x} = f(x) \mid \dot{\vec{x}} = \vec{f}(\vec{x}, \vec{y})$$

Rule 23 (Diff. eq. (one variable)).

$$\frac{\forall\, s \in \mathbb{R}_{\geq 0} \bullet \boldsymbol{\kappa}_s = \boldsymbol{\kappa}_0 + \int_0^s f(\boldsymbol{\kappa}_t)\, dt}{\dot{x} = f(x) \xrightarrow{t, x = \boldsymbol{\kappa}_0, x := \boldsymbol{\kappa}_t} \dot{x} = f(x)}$$

Rule 24 (Diff. eq. (multiple variables)).

$$\frac{\forall\, s \in \mathbb{R}_{\geq 0} \bullet \vec{\boldsymbol{\kappa}}_s = \vec{\boldsymbol{\kappa}}_0 + \int_0^s \vec{f}(\vec{\boldsymbol{\kappa}}_t, \vec{\psi})\, dt}{\dot{\vec{x}} = \vec{f}(\vec{x}, \vec{y}) \xrightarrow{t, \vec{y} = \vec{\psi} \wedge \vec{x} = \vec{\boldsymbol{\kappa}}_0, \vec{x} := \vec{\boldsymbol{\kappa}}_t} \dot{\vec{x}} = \vec{f}(\vec{x}, \vec{y})}$$

Notation (including lifted operators on lists):

$\langle \kappa_1, \ldots, \kappa_n \rangle$ a list $\vec{\kappa}$ where $\#\vec{\kappa} = n$

$$\vec{\kappa} + \vec{\psi} = \langle (\kappa_1 + \psi_1), \ldots, (\kappa_n + \psi_n) \rangle \qquad \text{provided } \#\vec{\kappa} = \#\vec{\psi} = n$$

$$\int_a^b \vec{f}(\vec{\kappa}) = \left\langle \int_a^b f_1(\vec{\kappa}), \ldots, \int_a^b f_n(\vec{\kappa}) \right\rangle \qquad \text{provided } \#\vec{f} = n$$

Figure 18: Semantics (hybrid system)

of $\boldsymbol{\kappa}_0$. A solution $\boldsymbol{\kappa}$ gives a value for x at every time point as it evolves according to $\dot{x} = f(x)$, and is derived by calculating the integral of f from 0 to each time point. The guard on the transition, $x = \boldsymbol{\kappa}_0$, is valid only if the chosen solution starts with the current value for x in the environment, and the update, $x := \boldsymbol{\kappa}_t$, sets x to the final value of x having evolved according to $f(x)$ for t time units.

For instance, if $\boldsymbol{\kappa}$ is a solution for $\dot{x} = 1$ where $\boldsymbol{\kappa}_0 = 0$, then $\boldsymbol{\kappa}$ satisfies $\boldsymbol{\kappa}_t = t$ for all $t \in \mathbb{R}_{\geq 0}$, or in general for any starting time n, $\boldsymbol{\kappa}_t = t + n$. The transitions generated by the process $\dot{x} = 1$ are therefore of the following form.

$$\dot{x} = 1 \xrightarrow{t, x = n, x := n + t} \dot{x} = 1$$

Rule 24 generalises to multiple variables $\vec{x}$ controlled concurrently by differential equations, which may depend on $\vec{x}$ and other variables $\vec{y}$. These other variables must not be continuous, and hence their initial value is fixed at $\vec{\psi}$ and this value remains constant throughout the duration t. To construct the solution $\vec{\boldsymbol{\kappa}}$ for multiple equations requires straightforward lifting of operators to lists of values, as shown in Fig. 18. The transitions in Rules 23 and 24 use only the initial and final values of $\boldsymbol{\kappa}$, which is sufficient under the well-formedness constraint that each variable is controlled by at most one differential equation process. This constraint may be relaxed by using a more general label type, $(t, x, \boldsymbol{\kappa})$, from which the required guard and update may be extracted, and parallel processes modifying x must agree on its value at all intermediate time points in $\boldsymbol{\kappa}$.

7. Conclusions

This paper has used basic notions for describing concurrent systems from process algebras extended with local variables, to formalise both a classic neural network and a more recent biologically-inspired neural network which operates in real time. The motivation for the latter formalisation was to better understand the details of neurobiology, in terms of individual neurons and network dynamics. Undertaking the formalisation of the model in [10] was nontrivial, with some details emerging only after many readings of the natural language, mathematical equations, and a page of simulation code. Some notable causes of difficulty were that the simulation code is a discretised version (time steps of 1ms) of the real-time behaviour of the system, and for reasons of efficiency contains some data structures and code than had no direct correlates in the mathematical description. This appears to be typical for many presentations of systems and simulation code in the literature.

The behaviour of a process is described in terms of its trace, the sequence of atomic, visible actions it takes. The trace of the classic neural networks generates a sequence of events which are parameterised by values corresponding the activations of the individual neurons, while the behaviour of the real-time network is the spike train, i.e., the timing of spikes of individual neurons. This correspondence between the formal model, its behaviour, and the mathematical underpinnings of artificial neural network research is important for validating models against theories and experimental data.

The process algebra for the classic neural network mixes state-based activity with typical event-based communication from process algebras. This helped to combine the basic mathematical/algorithmic foundation of neural networks with a more abstract description of how neurons interact. The adaptation to hybrid behaviour involved extending the semantics to include a new label type specifying a delay, and operators that controlled delay durations and the changes in variables over time. Because an operational semantics using labels was used, the hybrid syntax extensions straightforwardly extend the semantics and preserved the original rules (i.e., the semantics is *modular*, as defined by Mosses [51]). The intention is that the language serve as the basis for formalising other neural network models, and if the language needs further extension these can be made with minimal disruption to existing rules. The language may of course be used for specifying other types of hybrid systems as well.

With the existing practice in the neural network literature of describing systems semi-formally (a mixture of natural language and mathematical equations) there is a large gap between those descriptions and their eventual "implementations" as simulation code, typically in a programming language such as MATLAB [52]. Since the process language is formalised, there is the potential for directly simulating the specifications given in this paper [53, 54] or after translation into hybrid automata [55, 56], although there is likely to be a large computational overhead for simulating neural networks directly when compared to lower level purpose-built simulation code. Such simulation code may be developed formally based on the semantics [57, 58]. Another advantage of the formalisation is in local analysis [59]. The models can hence be used to show that the model simulated is the model described, and thus achieve a more rigorous level of validation. An alternative possibility is using the models in this paper as the end point of some more abstract specification [60, 61], however given their intentionally imprecise behaviour and complex emergent behaviour, it is unclear if this is in general possible or even desirable for real neural networks.

One of the advantages of formalising neural networks beyond obtaining a deeper and more complete understanding of its behaviour is to help in comparing models using different structures. Differences in assumptions about communication between neurons or the local effect of spikes can be stated more clearly. Of particular interest is in the integration of different facets of neural activity, for instance, extending the foundation model to exhibit properties of working memory [4], to incorporate the creation of new neurons (neurogenesis [62, 63]), or adding a mechanism by which the network can tell time based on properties of individual neurons [64] or on the interactions of the network [65].

References

[1] W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics 5 (1943) 115–133.

[2] F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain., Psychological Review 65 (1958) 386–408.

[3] E. Izhikevich, J. Gally, G. Edelman, Spike-timing dynamics of neuronal groups, Cerebral Cortex 14 (2004) 933–944.

[4] G. Mongillo, O. Barak, M. Tsodyks, Synaptic theory of working memory, Science 319 (2008) 1543–1546.

[5] T. Natschlger, W. Maass, Spiking neurons and the induction of finite state machines, Theoretical Computer Science 287 (2002) 251 – 265.

[6] W. Maass, H. Markram, On the computational power of circuits of spiking neurons, Journal of Computer and System Sciences 69 (2004) 593 – 616.

[7] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[8] R. Milner, A Calculus of Communicating Systems, Springer-Verlag New York, Inc., 1982.

[9] J. A. Bergstra, J. W. Klop, Process algebra for synchronous communication, Information and Control 60 (1984) 109–137.

[10] E. M. Izhikevich, Polychronization: Computation with spikes, Neural Computation 18 (2006) 245–282.

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

[12] J. Fisher, T. A. Henzinger, Executable cell biology, Nature Biotechechnology 25 (2007) 1239–1249.

[13] G. Bernot, J.-P. Comet, A. Richard, J. Guespin, Application of formal methods to biological regulatory networks: extending Thomas' asynchronous logical approach with temporal logic, Journal of Theoretical Biology 229 (2004) 339 – 347.

[14] L. Cardelli, Abstract machines of systems biology, in: C. Priami, E. Merelli, P. Gonzalez, A. Omicini (Eds.), Transactions on Computational Systems Biology III, volume 3737 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 145–168.

[15] L. Calzone, N. Chabrier-Rivier, F. Fages, S. Soliman, Machine learning biochemical networks from temporal logic properties, in: C. Priami, G. Plotkin (Eds.), Transactions on Computational Systems Biology VI, volume 4220 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2006, pp. 68–94.
[16] F. Ciocchetta, J. Hillston, Bio-PEPA: A framework for the modelling and analysis of biological systems, Theoretical Computer Science 410 (2009) 3065 – 3084.
[17] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, S. Tini, An overview on operational semantics in membrane computing, International Journal of Foundations of Computer Science 22 (2011) 119–131.
[18] H. Kugler, A. Larjo, D. Harel, Biocharts: a visual formalism for complex biological systems, Journal of The Royal Society Interface 7 (2010) 1015–1024.
[19] R. Donaldson, M. Calder, Modular modelling of signalling pathways and their cross-talk, Theoretical Computer Science 456 (2012) 30 – 50.
[20] R. Ruknas, J. Back, P. Curzon, A. Blandford, Verification-guided modelling of salience and cognitive load, Formal Aspects of Computing 21 (2009) 541–569.
[21] L. Su, H. Bowman, P. Barnard, B. Wyble, Process algebraic modelling of attentional capture and human electrophysiology in interactive systems, Formal Aspects of Computing 21 (2009) 513–539.
[22] L. Smith, A framework for neural net specification, IEEE Transactions on Software Engineering 18 (1992) 601–612.
[23] P. Machado, S. Meira, On the use of formal specifications in the design and simulation of artificial neural networks, in: J. Bowen, M. Hinchey (Eds.), ZUM '95: The Z Formal Specification Notation, volume 967 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1995, pp. 63–82.
[24] A. Senyard, E. Kazmierczak, L. Sterling, Software engineering methods for neural networks, in: Software Engineering Conference, 2003. Tenth Asia-Pacific, IEEE, pp. 468–477.
[25] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, S. Tini, Compositional semantics of spiking neural P systems, The Journal of Logic and Algebraic Programming 79 (2010) 304 – 316.
[26] G. Dorffner, H. Wiklicky, E. Prem, Formal neural network specification and its implications on standardization, Computer Standards & Interfaces 20 (1999) 333 – 347.
[27] E. Fiesler, Neural network classification and formalization, Computer Standards & Interfaces 16 (1994) 231 – 239.
[28] I. D. Zaharakis, A. D. Kameas, Modeling spiking neural networks, Theoretical Computer Science 395 (2008) 57 – 76.
[29] J. Bergstra, C. Middelburg, Process algebra for hybrid systems, Theoretical Computer Science 335 (2005) 215 – 280.
[30] P. Cuijpers, M. Reniers, Hybrid process algebra, The Journal of Logic and Algebraic Programming 62 (2005) 191 – 245.
[31] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, R. R. H. Schiffelers, Syntax and consistent equation semantics of hybridχ, J. Log. Algebr. Program. 68 (2006) 129–210.
[32] C. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1995.
[33] C. Morgan, The specification statement, ACM Trans. Program. Lang. Syst. 10 (1988) 403–419.
[34] R. J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Springer-Verlag, 1998.
[35] J. M. Spivey, The Z Notation: A Reference Manual, Prentice Hall, second edition, 1992.
[36] D. Rumelhart, G. Hinton, R. Williams, Learning representations by back-propagating errors, Nature 323 (1986) 533–536.
[37] R. Colvin, I. J. Hayes, CSP with hierarchical state, in: M. Leuschel, H. Wehrheim (Eds.), Integrated Formal Methods (IFM 2009), volume 5423 of *Lecture Notes in Comp. Sci.*, Springer, 2009, pp. 118–135.
[38] R. J. Colvin, I. J. Hayes, A semantics for Behavior Trees using CSP with specification commands, Science of Computer Programming 76 (2011) 891–914.
[39] J. L. Elman, Finding structure in time, Cognitive Science 14 (1990) 179 – 211.
[40] E. Izhikevich, Simple model of spiking neurons, Neural Networks, IEEE Transactions on 14 (2003) 1569–1572.
[41] E. Izhikevich, Which model to use for cortical spiking neurons?, Neural Networks, IEEE Transactions on 15 (2004) 1063–1070.
[42] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183 – 235.
[43] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets, volume 3098 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 87–124.
[44] X. Nicollin, J. Sifakis, An overview and synthesis on timed process algebras, in: K. G. Larsen, A. Skou (Eds.), Computer Aided Verification (CAV), volume 575 of *Lecture Notes in Computer Science*, Springer, 1992, pp. 376–398.
[45] S. Schneider, Concurrent and Real-time Systems: The CSP Approach, Wiley, 2000.
[46] S. Song, K. Miller, L. Abbott, Competitive Hebbian learning through spike-timing-dependent synaptic plasticity, Nature Neuroscience 3 (2000) 919–926.
[47] E. M. Izhikevich, Solving the distal reward problem through linkage of STDP and dopamine signaling, Cerebral Cortex 17 (2007) 2443–2452.
[48] H. Markram, Y. Wang, M. Tsodyks, Differential signaling via the same axon of neocortical pyramidal neurons, Proceedings of the National Academy of Sciences 95 (1998) 5323–5328.
[49] R. J. Colvin, I. J. Hayes, Structural operational semantics through context-dependent behaviour, Journal of Logic and Algebraic Programming 80 (2011) 392 – 426.
[50] W. Klimesch, EEG alpha and theta oscillations reflect cognitive and memory performance: a review and analysis, Brain Research Reviews 29 (1999) 169 – 195.
[51] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 195–228.
[52] MATLAB, version 7.12.0 (R2011a), The MathWorks Inc., 2011.
[53] P. C. lveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theoretical Computer Science 285 (2002) 359 – 405.
[54] F. Ciocchetta, A. Duguid, S. Gilmore, M. Guerriero, J. Hillston, The Bio-PEPA tool suite, in: Quantitative Evaluation of Systems (QEST), pp. 309 –310.
[55] T. A. Henzinger, P.-H. Ho, H. Wong-Toi, HyTech: A model checker for hybrid systems, in: O. Grumberg (Ed.), Computer Aided Verification (CAV), volume 1254 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 460–463.

[56] G. Frehse, PHAVer: algorithmic verification of hybrid systems past HyTech, in: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control, volume 3414 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 258–273.

[57] R. Alur, R. Grosu, I. Lee, O. Sokolsky, Compositional refinement for hierarchical hybrid systems, in: M. Benedetto, A. Sangiovanni-Vincentelli (Eds.), Hybrid Systems: Computation and Control, volume 2034 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2001, pp. 33–48.

[58] S. Ratschan, Z. She, Safety verification of hybrid systems by constraint propagation based abstraction refinement, in: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control, volume 3414 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 573–589.

[59] A. Platzer, J.-D. Quesel, KeYmaera: A hybrid theorem prover for hybrid systems (system description), in: A. Armando, P. Baumgartner, G. Dowek (Eds.), Automated Reasoning, volume 5195 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 171–178.

[60] J. Sanders, G. Smith, Emergence and refinement, Formal Aspects of Computing 24 (2012) 45–65.

[61] S. Stepney, F. Polack, H. Turner, Engineering emergence, in: Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on, pp. 89–97.

[62] M. Murphy, K. Reid, D. Hilton, P. Bartlett, Generation of sensory neurons is stimulated by leukemia inhibitory factor, Proceedings of the National Academy of Sciences 88 (1991) 3498–3501.

[63] J. Aimone, J. Wiles, F. Gage, Computational influence of adult neurogenesis on memory encoding, Neuron 61 (2009) 187.

[64] D. Buonomano, M. Merzenich, Temporal information transformed into a spatial code by a neural network with realistic properties, Science 267 (1995) 1028–1030.

[65] M. Matell, W. Meck, Cortico-striatal circuits and interval timing: Coincidence detection of oscillatory processes, Cognitive Brain Research 21 (2004) 139–170.