Using a Framework to Specify a Network of Temporal Neurons.

Leslie S. Smith
Centre for Cognitive and Computational Neuroscience
Department of Computer Science
University of Stirling, Stirling FK9 4LA, Scotland
telephone (44) 1786 467435 fax (44) 1786 464551 email: lss@cs.stir.ac.uk

Abstract

We discuss the use of frameworks (or formal models) for networks of temporal neurons, that is, neural networks using neurons in which precise signal timing matters. After discussing why one might require a framework at all, we review existing frameworks, and discuss the limitations of existing frameworks for their application to this more general form of neural network. With the aid of an example (a recurrent network of integrate-and-fire neurons) we show how one framework can be applied to this general form of neural network.

1 Background: who needs frameworks or formal models for neural networks?

Whether one believes that neural networks need frameworks for their specification or not depends on what one believes a neural network is. If one's view of a neural network is that it operates using one of a small number of standard algorithms (for example it is either a simple backpropagation network, or a Kohonen network, or a Hopfield network), then investing energy in producing frameworks or formal models seems wasteful. On the other hand, if one sees neural networks as a broader discipline, going from the simulation of biologically realistic neural networks, through to modelling parallel systems which communicate by passing scalar values (such as the network types mentioned above) then a framework for expressing the specification becomes more important. Further, if one wishes to consider networks in which the precise timing of the signals passing between processing elements matters then formal frameworks for their specification become more important.

So, who *does* need frameworks or formal models? Neural networks are viewed by different groups in different ways: for example, they are seen as

a form of pattern recognition system This is the view of neural networks commonly found amongst those developing them for industrial and commercial applications. Often these people believe that this is the only viewpoint!

a (perhaps idealized) model of a real neural system This view is more commonly found amongst computational neurobiologists. They are attempting to simulate parts of real neural systems to see if the simulation outcome follows that of their theories.

a form of parallel computer architecture This view is found amongst those who are trying to see how Neural Networks fit in with other developments in Computing. It is also found amongst some proponents of VLSI implementations - particularly digital VLSI Neural Network implementations.

a form of model/parameter computing system This view is common amongst those who have come to Neural Networks from other branches of pattern recognition, and who see the architecture choice as model selection, and weight manipulation as parameter estimation.

and there are other valid viewpoints as well. For the development of the field, it is important that people holding differing views talk to (rather than at) each other. Frameworks for neural networks which are not grounded in any one view of the field can provide a method for allowing different researchers to interact.

In this paper, section 2 discusses existing frameworks, and how they cope with recurrent networks of time-based neurons. Section 3 describes one framework in detail, then uses it to specify a particular network of temporal neurons, namely a recurrent network of integrate-and-fire neurons.

2 Models, frameworks, and formalizations for neural networks.

Given the above, it is hardly surprising that there is a body of work on frameworks for neural networks. However, many of these frameworks are aimed at specific aspects of neural networks, rather than being general. The nature of the formal models and frameworks in the literature varies very considerably: primarily they are concerned with description of a network, rather than with the proof of fundamental theorems about classes of networks, the only exception being Golden [12]. He produces a statistical framework for neural nets by seeking Maximum A Posteriori (MAP) estimates from the dynamical system made up of a neural network and its learning rule under the influence of a training regime. He evaluates these MAP estimates for a range of standard neural network types. However, we are still a long way from being able to develop theorems about the behaviour of general adaptive neural networks that allow their emergent properties to be discovered. In this paper, we will restrict ourselves to descriptive frameworks, though we will briefly comment on formal proofs at the end.

Descriptive frameworks range from programming-language based to equational, from frameworks attempting to unify a number of existing neural network approaches to those which attempt to be much more general, from those concerned purely with neural networks to those which are formal models of general systems with many interacting entities. The earliest work

on this area appears to date from the *nEuro 88* conference in Paris: four papers [24, 16, 2, 5] presenting different frameworks or formal models were published and these led on to further work ([25, 13, 27, 3] respectively).

2.1 A taxonomy of descriptive frameworks.

The descriptive frameworks in the literature fall into four categories:

languages for describing real neural networks There are a number of simulation packages which are intended for modelling real neural systems, such as SWIM [8], and there are packages which are intended for general continuous system modelling, such as SABER [4] or SPICE [28]. These can be used to model any continuous system, and this includes modelling neurons or neural networks.

languages for describing artificial neural networks AXON [13] is primarily concerned with defining layers (or *slabs*, in AXON parlance), connections between slabs, the processing element function, and the order of updating of the slabs. Other language based techniques such as CONNECT [17] are concerned with providing a very-high level language from which C++ will be generated, or are systems for programming neural network simulation systems, such as nC or N [27]. Many other simulation packages have special-purpose languages for describing the network to be simulated to the simulator. ANSpec [23] and NACRE [6] are both special-purpose languages based on ACTORS [14], and again, are used in specific simulation systems. MENTAL [3] and MIND [18] are CSP-based [15] and use an occam-like syntax to describe neural networks. Both are intended as specification techniques for a broad range of implementations. Dorffner et al [7] develop a C++ like syntax for describing networks, their environment, and the input/output interface between them.

mathematical notations for any form of neural network Both Fiesler and Caulfield [10] and Smith [25] develop a formal neural network specification using a mathematical notation. Fiesler and Caulfield's notation is oriented towards existing algorithms for artificial neural networks, whereas Smith's is more general.

general formal models for specifying parallel systems There are three formal models for parallel system which have been influential on formal models for neural networks. Hewitt's ACTORS [14] has influenced the design of ANSpec, and both Hoare's CSP [15] and Milner's CCS [22] influenced Smith [25] and Dorffner [7].

Given this array of techniques, how can one choose which are the most appropriate? If one wants to work with neurons at the level of ionic channels, or compartments, then SWIM or SABER are best. On the other hand, if one is simply attempting to define a network to a particular simulator, one needs to use the language defined by that simulator. However, if one would like the same network to be implemented on a different simulator, or programmed up in a general programming language, one would like to have some more general form of formal description. For this, it would seem as though AXON, Nc, CONNECT or ANSpec would be

appropriate. CONNECT has problems with defining the precise dynamics of a simulation, and so it should be ruled out.

If one wishes to simulate one of the network types which requires the interpretation of patterns of neuron outputs over time (as for example in Abeles synfire chains [1], or McGregor's sequential configuration model [21]), then most of the above frameworks would need considerable extension. Indeed, language based approaches require some form of semantics for considering accurately synchronisation and near–synchronisation between neural outputs, and few existing neural network specification languages permit this. The difficulties become even more visible if one is considering temporal neurons, as may be required for either a hardware implementation, or a realistic neural network simulation. One then needs to consider the neural network as a set of interacting entities: indeed, one may wish to consider each neuron as a set of interacting entities. For this, one needs something more powerful than these languages for defining neural networks. What we seem to want is something that has some of the facilities more often associated with real neuron simulation, but with abstraction facilities permitting work at a higher level as well.

It can be argued that one should use one of the very general techniques for specifying parallel systems. These provide an additional advantage: they can be hierarchical. This is useful if one wishes to develop a network by specifying the neurons as being composed of functional parts, and then building up the network hierarchically. Timing can still present problems: interaction between entities in ACTORS and CSP is event-based, so that although the order of signals between entities may be well specified, their exact timing may not be.

Restating the above question: if one wants to specify a neural network which is recurrent, and whose units are temporal neurons, and if one would like the specification not to be tied to any single simulation package - perhaps because it may be implemented in hardware at some point - how should one set about specifying the network?

2.2 Which framework should one use for networks of temporal neurons.

Most of the frameworks discussed above were not designed with neurons which exhibit any behaviour over time. The primary exceptions are those frameworks intended for modelling real neurons: these (SABER, SWIM, and SPICE) are tools for simulation which cope well with time. However, they are not really frameworks for artificial neural networks at all, but simulation tools. ACTOR-based systems (NACRE and ANSpec, MENTAL, and MIND) and CSP-based systems (MENTAL, and MIND) can usefully specify event ordering, but not the precise timing of events. Fiesler's approach does not mention time at all.

The two frameworks which do discuss precise timing are those of Dorffner et al, and Smith. Dorffner's NSpec specification language has *signals* whose timing can be used at neurons. Smith shows how his specification technique can be used to model systems with continuous dynamics: that is, including (real) time in the specification. In what follows, we will use the framework advanced by Smith, as it is illustrative of the problems involved, but powerful enough to allow expression of time-based networks. The author believes that Doffner et al's approach could also be used, but that the other approaches described in section 2.1 would

need considerable extension.

3 Specifying a network of integrate-and-fire neurons

3.1 Integrate-and-fire neurons.

In this section, we will specify a network of interconnected integrate-and-fire neurons. Such neurons have been used for slightly more realistic neural simulation for a long time, and are well described in [11]. Their power has been explored by Maass [19, 20], and applied, for example by Smith [26]. The essential characteristics of an integrate-and-fire neuron are as follows:

Each neuron has a number of inputs. Each input is weighted, and the weighted input alters the internal (voltage–like) activation. At the same time, the activation is leaking away towards 0. If the activation reaches some threshold, the neuron outputs a pulse and resets the activation to 0. After this, it may completely ignore its input for some time (the *absolute refractory period*) and possibly reduce its sensitivity to input for some further time (the *relative refractory period*).

3.2 The framework.

We will use the framework set out by Smith [25]. For completeness, we restate the framework here.

Informally, a network consists of *nodes* and directed *arcs* connecting pairs of *ports* each associated with a node. The network exists within an *environment*, which is treated as a special case of a node. With the exception of the environment, nodes are described by instantiation of a *generator*.

3.2.1 The specification technique

More formally, we start from a number of sets, from which we will build up the notation. These sets are

NODENAMES: the set of names for nodes. (e.g. strings of bounded length).
GENERATORNAMES: the set of names for generators.
PORTS: the set of ports on nodes.
STATESPACES: the set of possible state spaces for nodes.
FUNCTIONS: the set of functions that a node may implement.

The set PORTS consists of two disjoint subsets, INPORTS and OUTPORTS, corresponding to the direction of information flow at that port.

All input to and output from the network is from or to the environment, E. The environment has ports $p(E) \subset$ PORTS, and this set consists of input ports $p_{in}(E) \subset$ INPORTS and output ports $p_{out}(E) \subset$ OUTPORTS. Nothing else about the environment is defined.

We now define generators and nodes. Logically, one should start with the generators, since nodes are generated from these. However, in a network, the fundamental entities are the nodes: though the generator concept allows the construction of a number of nodes from a common startpoint, as well as allowing hierarchical net construction, it is secondary. We therefore start with the node. It is defined as a tuple:

$$n \in \text{ NODENAMES } \times set(\text{PORTS}) \times \text{ STATESPACES } \times \text{ FUNCTIONS} \tag{1}$$

where set(PORTS) is the set of subsets of PORTS. We write

$$n = (\text{name}(n), p(n), \text{STATE}(n), F_n) \tag{2}$$

where name(n) defines the node's name, and $p(n) = p_{in}(n) \cup p_{out}(n)$, as with the environment, defines the set of ports. We will write $p^i_{in}(n)$ or $p^i_{out}(n)$ for each port. STATE(n) is the set of possible or reachable states for node n. F_n defines the function of the node, that is, how the outputs (i.e. values placed on output ports) are computed, and how the state updates. Where it is useful, we separate out the state update part of F_n, calling it F^{int}_n, from the port output part, calling it F^{out}_n.

Nodes exist over time: indeed, they may evolve over time. The name of the node and the ports do not change, but the state, the values on the ports, and possibly the function do change. We write state$(n, t) \in$ STATE(n) for the node state at time $t \geq 0$, state$(n, 0)$ being the node's initial state, and we write $p^i_{in}(n, t)$ (or $p^i_{out}(n, t)$) for the value on port $p^i_{in}(n)$ (or $p^i_{out}(n)$) at time t. Where the node to which the port belongs is clear, we drop the parameter n. Since the function F_n may adapt, we write $F_{n,t}$ for the function implemented by the node at time t. This defines the values to be placed on the $p_{out}(n)$ ports at time t, and how the state is updated at time t. This will depend on the initial function, $F_{n,0}$, on what has been received on the $p_{in}(n)$ ports up to time t, and on the initial state of the node.

Arcs are defined as pairs:

$$a \in \text{OUTPORTS} \times \text{INPORTS} \tag{3}$$

Thus, each directed arc a joins one output port to one input port. Thus, if $a = (f_a, t_a)$, writing N for the set of nodes,

$$f_a \in \bigcup_{r \in N \cup E} p_{out}(r)$$

$$t_a \in \bigcup_{r \in N \cup E} p_{in}(r)$$

Writing A for the set of arcs, we can characterise the net, $\mathbf{N}$, itself as

$$\mathbf{N} = (N, A) \tag{4}$$

Each node, n, is generated from a generator, g. The aim of introducing this secondary entity is twofold: firstly, it provides a common startpoint for a number of nodes, and secondly, by

providing a method for producing a generator from a complete network, it allows networks to be built up hierarchically. The generator must be able to define the name, ports and statespace of the node, the initial state and initial function, and how these will evolve in time. We have taken a parameter based approach: the generator provides a template for the name and function of the node, and these are precisely defined using parameters. Each g is a triple,

$$g = (\text{gname}(g), \text{PAR}(g), F_g) \tag{5}$$

where $\text{gname}(g) \in \text{GENERATORNAMES}$, $\text{PAR}(g)$ is the parameter space for this generator, and F_g is a template for the function F_n. Again, where this is useful, we may split this into F_g^{int} and F_g^{out} as for F_n. This is not completely general: however, the functions used in nodes are usually relatively simple, and generally fall into a small number of classes. We will write G for the set of generators. The parameter space is used to specify all the other things that need to be specified: that is, how name(n) is derived from gname(g), what the set $p(n)$ should be, what STATE(n) should be, how F_n should be derived from F_g, what state(n,0) and $F_{n,0}$ should be, and how they should evolve in time. This is accomplished in two steps: firstly the actual parameters of g are set:

$$\text{setparameters}(g) = (\text{gname}(g), \text{par}(g), F_g) \tag{6}$$

(simply choosing $\text{par}(g) \in \text{PAR}(g)$) and then the generator with its parameters set is instantiated:

$$\text{instantiate}(\text{gname}(g), \text{par}(g), F_g) = n = (\text{name}(n), p(n), \text{STATE}(n), F_n) \tag{7}$$

This defines the name of the node n, its ports, its state space, and function. It also implicitly sets up state$(n, 0)$ and $F_{n,0}$ since these are defined by the selection of the parameters. We have not precisely formalised the generation of STATE(n) from the parameters. In general, we will use a subspace of PAR(g), one spanned by some of the parameters. The initial state, state$(n, 0)$, will be defined by the actual value of these parameters.

This framework can be applied to many different types of interacting entities: clearly it is the form of the elements (as defined by the g, and the mappings instantiate and setparameters) and the pattern of interconnection which make the framework produce something which is recognisably a neural net. Thus, for example, we sometimes identify part of the internal state with the weights of a neuron, or with an activation level.

The restriction that each port may be an endpoint of at most one arc seems odd at first: we are used to axonic outputs going to many other neurons. However, the arcs do not represent either axonic or dendritic links; they are simply instant communication paths. All the active elements including synapses will be contained in the nodes of the net. This restriction allows us to make all the links identical whereas alternative approaches would require us to characterise links as well as nodes. This approach permits both subdivision of each neural element (so that nodes may be parts or compartments of a neuron) and clustering of neural elements (so that nodes may be networks of neurons).

3.2.2 Using the specification technique hierarchically

Nets can be defined hierarchically by forming a generator from a whole net, and then instantiating this as a node. Considering the net

$$\mathbf{N} = (N, A)$$

we need to produce a generator for the new node which will replace $\mathbf{N}$:

$$g = g(\mathbf{N}) = (\text{gname}(g(\mathbf{N})), \text{PAR}(g(\mathbf{N})), F_{g(\mathbf{N})}) \tag{8}$$

To produce the generator $g(\mathbf{N})$ entails selecting a new name $\text{gname}(g(\mathbf{N}))$, defining the parameter space $\text{PAR}(g(\mathbf{N}))$, and defining a template function $F_{g(\mathbf{N})}$. The name can be chosen from GENERATORNAMES, but the parameter space and template function must be constructed. The parameter space can be constructed (for example) by considering its elements to have the form

$$(s_0, s_1, \ldots, s_r) : s_i \in S_i$$

where S_i is the range of parameters for the i'th parameter. We can use s_0 and s_1 to define the ports. These come from the arcs of t e original net, specifically, from the subsets of A:

$$A_O = \{a \in A : t_a \in p_{in}(E)\} \text{ and } A_I = \{a \in A : f_a \in p_{out}(E)\} \tag{9}$$

which are the arcs between the net $\mathbf{N}$ and its environment E. Their endpoints in $\mathbf{N}$ (i.e. f_a in A_O and t_a in A_I) will generate the ports of the node. Let s_0 parameterise the input ports, and s_1 the output ports. s_0 can be used to define how many input ports will be generated from each t_a in A_I, and s_1 how many output ports will be defined from each f_a in A_O. Note that some of the t_a and f_a may be left unused, while others may give rise to several ports. The parameters $s_2, \ldots, s_r$ parameterise everything we wish to be externally visible (i.e. parameterisable) from the parameters used in the construction of all the nodes of the original net. $F_{g(\mathbf{N})}$ is defined implicitly by the F_n where $n \in N$. Working in this way, the initial state of the new node nd the initial function may be determined by the initial states and initial functions of all the nodes in the original network, or the parameterisation used in setting up these states in the original nodes can be re-used in the setting up of the new node. This construction can be nested to any desired level; however, circular definitions must be avoided so that eventually all the nodes are defined in terms of elementary generators.

3.3 The example specfication

We will start by specifying the component parts of an integrate-and-fire neuron, then build a specification for a single integrate-and-fire neuron and then the specification for a network of these neurons.

3.3.1 Specifying the integrate-and-fire neuron

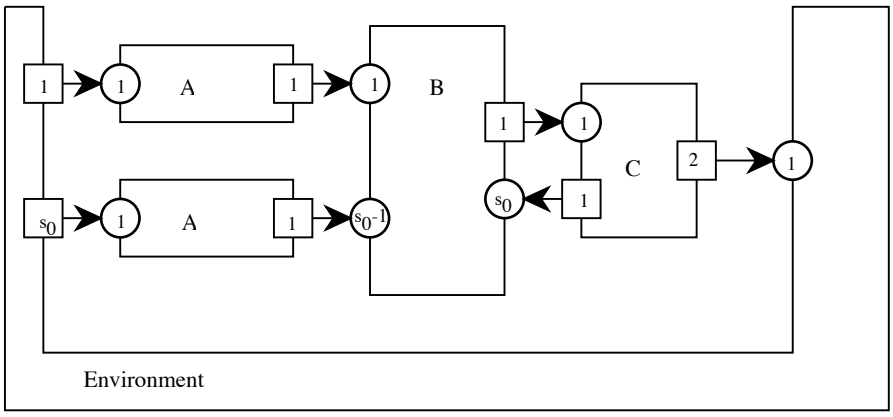

Figure 1: Arrangement and constituent elements of an integrate-and-fire neuron. A are synapse elements, B activation element, and C output element. Numbered squares are output ports, numbered circles input ports.

Figure 1 shows the component parts we have chosen. To specify a generator for the synapse element, we set

$$g_{\text{synapse}} = (\text{synapse}, \text{PAR}_{\text{synapse}}, F_{g_{\text{synapse}}}) \tag{10}$$

We then set $\text{PAR}_{\text{synapse}}$ up so that it contains the spaces for the parameters we may wish to use inside the synapse. These control both the formation of ports and the definition of the eventual functionality of the synapse. However, here, we can constrain the synapse to have one input and one output port, so we need only consider what form we might want the eventual function F_{synapse} to take. For this we will use the convolution function

$$F_{g_{\text{synapse}}}(t) = W \int_0^t C(x) p_{\text{in}}^1(t-x)dx \tag{11}$$

(where W is the weight of the synapse, $C(x)$ is a convolving function, and $p_{\text{in}}^1(t)$ is the input to the synapse) to be sufficiently general. Thus, $\text{PAR}_{\text{synapse}}$ is the space from which we can choose W and $C(x)$. For example

$$\text{PAR}_{\text{synapse}} = \{(s_0, s_1, W, C) : s_0 \in \{1\}, s_1 \in \{1\}, W \in \mathcal{R}, C \in L^2(\mathcal{R})\}$$

where $\mathcal{R}$ is the real numbers, and where s_0 and s_1 define the number of input and output ports. setparameters(g_{synapse}) will then consist of selecting W and $C(x)$, and instantiate of providing this instance of the generator called synapse with a unique name, setting up the two ports, and the initial state (in this case, null).

For the activation element, the situation is a little more complex. We start the same way:

$$g_{\text{activation}} = (\text{activation}, \text{PAR}_{\text{activation}}, F_{g_{\text{activation}}}) \tag{12}$$

The parameter space will need to define the number of associated synapses, as well as how the element will generate its output. For generality (and particularly because we do not want to decide whether we will eventually implement this in a digital or analogue form) we define

$F_{g_{\text{activation}}}$ implicitly as a differential equation:

$$\frac{dA}{dt} = \sum_{i=1}^{s_0-1} p^i_{\text{in}}(t) - DA \tag{13}$$

(where s_0 is one more than the number of synapses and D is the dissipation (see [11]) of the leaky integrator) subject to $A \geq A_{\min}$, the minimum permitted activation. $A(t)$ will be part of the state of the instantiated element, and the output, $p^1_{\text{out}}(t)$, is simply $A(t)$. In addition, we have a reset signal (which occurs when the neuron fires) from the last input port, $p^{s_0}_{in}$, and when this is set (i.e. is 1), A is reset to 0. This will be used to implement the refractory period. The time when the reset signal returns to 0 is recorded as T_{RPend}. This will also be part of the state of the instantiated element. To implement the relative refractory period, we replace the $p^i_{\text{in}}(t)$ in equation 13 by $\frac{t-T_{\text{RPend}}}{\text{RRP}} p^i_{\text{in}}(t)$ while $T_{\text{RPend}} \leq t \leq (T_{\text{RPend}} + \text{RRP})$. We can now describe $\text{PAR}_{\text{activation}}$:

$$\text{PAR}_{\text{activation}} = \{(s_0, s_1, A_{\min}, D, \text{RRP}) : s_0 \in \mathcal{I}_1, s_1 \in \{1\}, A_{\min}, D \in \mathcal{R}^+, \text{RRP} \in \mathcal{R}\} \tag{14}$$

where $\mathcal{I}_1$ is the positive integers and s_0 is the number of input ports ($s_0 - 1$ is the number of synapses, and $p^{s_0}_{in}$ is the reset input port). s_1 is the number of output ports (fixed at 1), $A_{\min}$ is the minimum permitted activation, D is the dissipation, and RRP the relative refractory period. setparameters($g_{\text{activation}}$) will then fix the values for $(s_0, s_1, A_{\min}, D, \text{RRP})$, and instantiate provide the instance with a unique name, set up the ports, and the initial state. In this case, there is a non-null state, consisting of the initial activation, A_{initial}, and a value for T_{RPend}. The former is likely to be set to 0, (or some other appropriate initial value), and the latter to some value such that $T_{\text{RPend}} + \text{RRP} < 0$ so that the neuron is not in its relative refractory period at $t = 0$.

For the output element, we have again

$$g_{\text{output}} = (\text{output}, \text{PAR}_{\text{output}}, F_{g_{\text{output}}}) \tag{15}$$

The parameter space has to define the number of input and output ports (1, and 2 respectively), the threshold and refractory period for the neuron, and the shape of the pulse output by the neuron. We can characterise the parameter space by

$$\begin{aligned} \text{PAR}_{\text{output}} \quad = \quad & \{(s_0, s_1, \phi, L_{\max}, \text{RP}, s) : s_0 \in \{1\}, s_1 \in \{2\}, \\ & \phi, \text{RP}, L_{\max} \in \mathcal{R}^+, s \in S \subset L^2([0, L_{\max}])\} \end{aligned} \tag{16}$$

so that the threshold ϕ, the refractory period, RP, and the maximum pulse length are positive real numbers, and S defines the set of possible pulse shapes on the interval $[0, L_{\max}]$. The two output ports have different outputs: p^1_{out} outputs a 1 for time RP starting when $p^1_{\text{in}}(t) \geq \phi$, and p^2_{out} outputs a pulse of shape s starting when $p^1_{\text{in}}(t) \geq \phi$. setparameters($g_{\text{output}}$) then selects valid values for $(s_0, s_1, \phi, L_{\max}, \text{RP}, s)$, and instantiate provides the instance with a unique name, and sets up the ports.

To combine these elements to produce an integrate-and-fire neuron, we require to instantiate elements by applying setparameters() and instantiate() to the appropriate generators, and to set up the arcs between the instantiated elements. In order to set up an integrate-and-fire neuron with M inputs, we (a) apply repeatedly (M times) instantiate $\circ$ setparameters to

g_{synapse}, setting the appropriate initial weights and convolving function, and setting up M different names, say synapse$i$, for $1 \leq i \leq M$, (b) apply instantiate $\circ$ setparameters once to $g_{\text{activation}}$, setting s_0 to $M+1$, and the other parameters in equation 14 appropriately, and setting the instantiated element's name to activate1, and (c) apply instantiate$\circ$setparameters once to g_{output}, setting the parameters in equation 16 appropriately and setting the instantiated element's name to output1. We then need to join these elements together by appropriate arcs. Writing $p^i_{\text{in}}(\text{xyz})$ for the i'th input port of an element named xyz, we create the arcs (see figure 1)

$$\begin{aligned} (p^1_{\text{out}}(\text{synapse}i) &\quad , \quad p^i_{in}(\text{activation1})) : 1 \leq i \leq M \\ (p^1_{\text{out}}(\text{activation1}) &\quad , \quad p^1_{in}(\text{output1})) \\ (p^1_{\text{out}}(\text{output1}) &\quad , \quad p^{M+1}_{\text{in}}(\text{activation1})) \end{aligned} \tag{17}$$

In addition, we need to define the arcs between the environment and the neuron: these will be

$$\begin{aligned} (p^i_{\text{out}}(\text{environment}) &\quad , \quad p^1_{\text{in}}(\text{synapse}i)), 1 \leq i \leq M \\ (p^2_{\text{out}}(\text{output1}) &\quad , \quad p^1_{\text{in}}(\text{environment})) \end{aligned} \tag{18}$$

This completes the specification of an integrate-and-fire neuron with M inputs.

In the construction of this specification we have made choices. For example, we could have chosen to have implemented both RP and RRP at the output element. This could have been achieved by setting $\phi(t) = \infty$ during the refractory period, then reducing $\phi(t)$ back to ϕ_{initial} gradually during the relative refractory period. This would remove the need for the arc from the output element to the activation element. The resulting integrate-and-fire neuron has differences from the one specified here, and one can use the specification technique to investigate these differences. The primary difference is that $p^i_{\text{in}}(\text{activation}), i = 1 \ldots s_0 - 1$ is ignored during the refractory period in the original specification, but is accumulated in the alternative one, so that the activation level at the start of the relative refractory period will be different in the two cases.

3.3.2 Specifying the network

To specify the whole network, we will need to produce two elements, an integrate-and-fire neuron, formed from the specification in section 3.3, and a delaying distributor element to be used for interconnecting the neurons. In addition, we will need to specify the arcs connecting these elements.

For the integrate-and-fire neuron, we start by setting

$$F_{g_{\text{IandF}}} = (\text{IandF}, \text{PAR}_{\text{IandF}}, F_{g_{\text{IandF}}}) \tag{19}$$

Following the technique outlined in section 3.2.2, we set up $\text{PAR}_{\text{IandF}}$ from the parameter spaces of the elements making up the neuron, and from the arcs. If the neuron is to have M inputs, then we will be able to define the ports by applying equation 9 to the ports defined in equation 17 and equation 18: that is, we will require M input ports, and one output port.

All the arcs in equation 17 will be internal to the generator of the neuron. The rest of the spaces in $\mathrm{PAR}_{\mathrm{IandF}}$ come directly from the parameter spaces of the elements of the neuron.

The elements of the neuron will be: M synapse elements, one activation element, and one output element. Thus,

$$\mathrm{PAR}_{\mathrm{IandF}} \subseteq \mathrm{PAR}_{\mathrm{synapse}}^{M} \times \mathrm{PAR}_{\mathrm{activation}} \times \mathrm{PAR}_{\mathrm{output}} \tag{20}$$

We can use a subspace without losing generality by noting that

$$s_0^{\mathrm{synapse}} \in \{1\} \text{ and } s_1^{\mathrm{synapse}} \in \{1\} \tag{21}$$

for all the synapses, and can be subsumed into the s_0^{IandF}. We also note that $s_0^{\mathrm{activation}} = M + 1 = s_0^{\mathrm{IandF}} + 1$, and thus does not need to be specified separately.

$$\begin{aligned}
\mathrm{PAR}_{\mathrm{IandF}} \quad = \quad & \{(s_0^{\mathrm{IandF}}, s_1^{\mathrm{IandF}}, \mathbf{W}, \mathbf{C}, s_1^{\mathrm{activation}}, A_{\min}, \\
& D, \mathrm{RRP}, s_0^{\mathrm{output}}, s_1^{\mathrm{output}}, \phi, L_{\max}, \mathrm{RP}, s) : \\
& s_0^{\mathrm{IandF}} \in \{M\}, s_1^{\mathrm{IandF}}, s_1^{\mathrm{activation}}, s_0^{\mathrm{output}} \in \{1\}, \\
& \mathbf{W} \in \mathcal{R}^M, \mathbf{C} \in L^2(\mathcal{R})^M, A_{\min}, D, \mathrm{RRP} \in \mathcal{R}, s_1^{\mathrm{output}} \in \{2\}, \\
& \phi, L_{\max}, \mathrm{RP} \in \mathcal{R}^+, s \in S\}
\end{aligned} \tag{22}$$

where s_0^{IandF} is the number of inputs to the neuron, s_1^{IandF} is the number of outputs of the neuron, $\mathbf{W}$ is the weight vector and $\mathbf{C}$ is vector of convolving functions associated with each neuron. $F_{g_{\mathrm{IandF}}}$ is defined by the composition of the $F_{\mathrm{synapse}}, F_{\mathrm{activation}}$ and F_{output}. As before, setparameters provides valued for these parameters, and instantiate provides the instance with a unique name, and sets up the initial state of the neuron.

For the delayed distributor element, we write

$$g_{\mathrm{delay}} = (\mathrm{delay}, \mathrm{PAR}_{\mathrm{delay}}, F_{g_{\mathrm{delay}}}) \tag{23}$$

The parameter space has to define the number of places the (single) input is to be transferred to, and the delay associated with each link. Thus we write

$$\mathrm{PAR}_{\mathrm{delay}} = \{(s_0, s_1, \mathbf{T}) : s_0 \in \{1\}, s_1 \in \mathcal{I}_1, \mathbf{T} \in \mathcal{R}^{s_1}\} \tag{24}$$

where $\mathbf{T} = (T^1, \ldots, T^{s_1})$ is the delay vector. The element function $F_{g_{\mathrm{delay}}}$ is a vector-valued function with elements p_{out}^i, each defined by

$$\begin{aligned}
p_{\mathrm{out}}^i(t) \quad &= \quad 0 \text{ for } 0 \leq t \leq T^i \\
&= \quad p_{\mathrm{in}}^1(t - T^i) \text{ for } t > T^i
\end{aligned} \tag{25}$$

setparameters selects the number of output ports, s_1, and the associated delay $\mathbf{T}$. instantiate provides a unique name.

To produce a network, we first set the parameters for and then instantiate the elements in pairs of IandF and delay elements. We arrange that there is an arc from the output of the IandF unit to the input of the delay element. The other arcs are specified will define the toplogy for the network. As an example, we will consider a network in which the environment

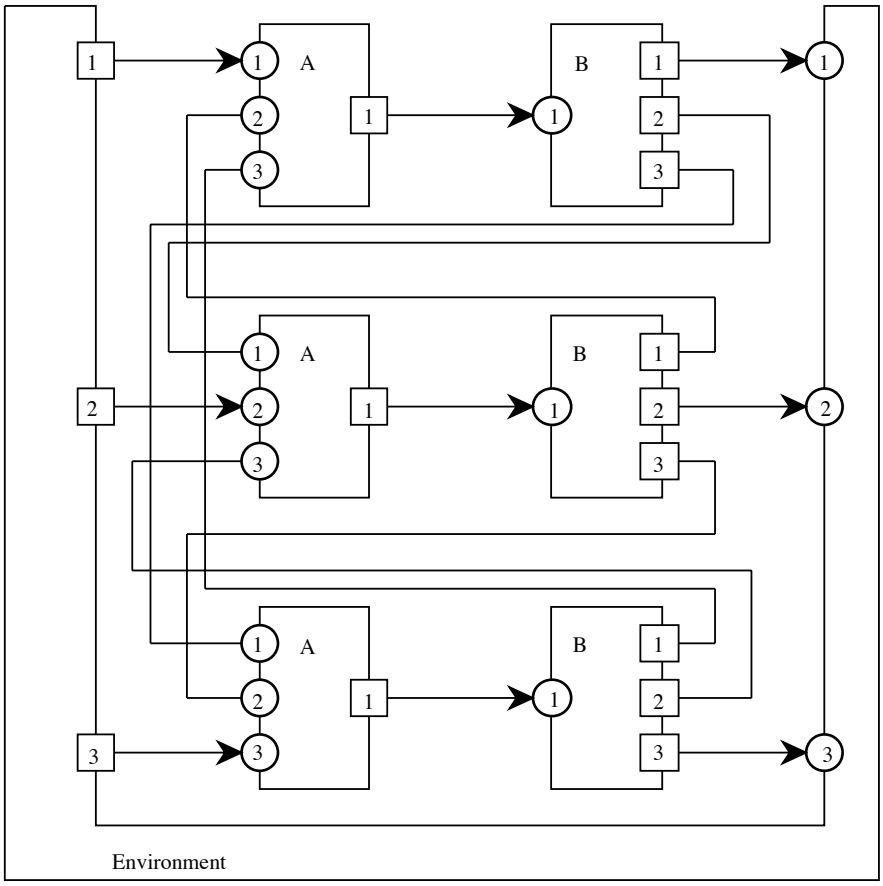

Figure 2: One possible network of 3 integrate-and-fire units. A's are integrate-and-fire units, B's are delay elements. Each delay element makes synapses with all the other integrate-and-fire units.

provides one input to each IandF unit, and each IandF unit has (delayed) synapses with the other units. This network is recurrent, and was the form of the neural network in [26]. For this network, we set the parameters for and then instantiate N neurons and delay elements, and give them names IandFi and delay$i$. The arc from the IandF unit to its delay element will be

$$(p^1_{\text{out}}(\text{IandF}i), p^1_{\text{in}}(\text{delay}i)) \tag{26}$$

The other internal arcs are

$$(p^j_{\text{out}}(\text{delay}i), p^i_{\text{in}}(\text{IandF}j), i, j \in \{1 \ldots N\}, i \neq j \tag{27}$$

and the arcs connecting to the environment are

$$(p^i_{\text{out}}(\text{environment}), p^i_{\text{in}}(\text{IandF}i)) \text{ and } (p^i_{\text{out}}(\text{delay}i), p^i_{\text{in}}(\text{environment})), i \in \{1 \ldots N\} \tag{28}$$

Figure 2 shows this network for $s_0^{\text{IandF}} = M = 3$.

The network specified is not adaptive: however this can be added straightforwardly. To maintain locality, information used in altering whatever adapts must be brought to the element

in which the adaptation occurs. Here, this would mean an arc from the output element of a neuron back to the synapse elements of that neuron so that Hebbian adaptation could be implemented. One could have a variety of different types of synapse: for example those receiving input from delay elements might be adaptive, and those receiving input from the environment not be adaptive. This could be implemented by using more than one generator, or by a more complex parameterisation of a single generator.

4 Conclusions and further work.

We have demonstrated that the framework in [26] can be used to specify a network of temporal neurons. This form of specification provides a method for describing neural networks without needing a decision about how they will eventually be implemented. The results of making different choices can be considered at specification time, and their effects on the system investigated. Although the framework has been used as a basis for a functional language implementation [9], current work is concentrating on the use of the framework purely for investigating the network in a formal way. The framework permits neural networks to be expressed formally in a mathematical notation. This helps to provide a basis for mathematical reasoning about neural network systems. The parameter space of the elements of the network, and hence of the network itself is expressed in the formalism, and this can be used to help define the range of parameters over which some assertion holds.

References

[1] M. Abeles. *Corticonics*. Cambridge University Press, 1991.

[2] B. Angéniol, J-Y. Le Texier, and J-B. Mateu. SLOGAN: an object–oriented language for neural network specification. In L. Personnaz and G. Dreyfus, editors, *Neural Networks from Models to Applications: nEuro 88 Conference Proceedings*, pages 641–652. IDSET, Paris, 1989.

[3] P. Bessiere, A. Chams, and P. Chol. MENTAL: A virtual machine approach to artificial neural networks programming. Technical report, ESPIRIT BRA Project 3049, 1991. Final Report.

[4] N.T. Carnevale, T.B. Woolf, and G.M. Shepherd. Neuron simulations with SABER. *Journal of Neuroscience Methods*, 33, 1990.

[5] P. Chol and T. Muntean. NEURAL: Towards an occam extension for neurocomputers. In L. Personnaz and G. Dreyfus, editors, *Neural Networks from Models to Applications: nEuro 88 Conference Proceedings*, pages 653–662. IDSET, Paris, 1989.

[6] B. Derot, P. Escande, and C. Moulinoux. NACRE: a neuron-oriented programming environment. In *Neuro-Nimes '89*, pages 183–199, 1989.

[7] G. Dorffner, H. Wiklicky, and E. Prem. Formal neural network specification and its implications on standardization. Technical Report OFAI TR-93-24, Austrian Research Institute for Artificial Intelligence, 1993.

[8] O. Ekeberg, M. Stensmo, and A. Lansner. SWIM - a simulator for real neural networks. Technical Report TRITA-NA-P9014, Royal Institute of Technology, Stockholm, 1990.

[9] J.E. Exton. A functional prototyping system for neural networks. Master's thesis, University of Stirling Depertment of Computer Science, March 1992.

[10] E. Fiesler and H.J. Caulfield. Neural network formalization. *Computer Standards and Interfaces*, 16(3):231–239, 1994.

[11] W. Gerstner. Time structure of the activity in neural network models. *Physical Review E*, 51(1):738–756, January 1995.

[12] R.M. Golden. A unified framework for connectionist systems. *Biological Cybernetics*, 59:109–120, 1988.

[13] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1990.

[14] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8, 1977.

[15] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[16] M. Kőhle and F. Schőnbauer. CONDELA – a language for neural networks. In L. Personnaz and G. Dreyfus, editors, *Neural Networks from Models to Applications: nEuro 88 Conference Proceedings*, pages 634–640. IDSET, Paris, 1989.

[17] G. Kock and N.B. Serbedzija. Artificial neural networks: from compact descriptions to C++. In M. Marinaro and P.G. Morasso, editors, *ICANN94: Proceedings of the International Conference on Artificial Neural Networks*, pages 1372–1375. Springer-Verlag, 1994.

[18] P. Koikkalainen. MIND: A specification formalism for neural networks. In T. Kohonen, K. Makisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 579–584. North-Holland, 1991.

[19] W. Maass. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8(1):1–40, 1996.

[20] W. Maass. On the computational power of noisy spiking neurons. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information processing Systems 8*, pages 211–217. MIT Press, 1996.

[21] R.J. MacGregor. *Theoretical Mechanics of Biological Neural Networks*. Academic Press, 1993.

[22] R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4), 1979.

[23] Science Applications International Corporation, San Diego, California. *ANSpec^{TM} User's Manual*, 1989.

[24] L.S. Smith. Formalizing neural networks. In L. Personnaz and G. Dreyfus, editors, *Neural Networks from Models to Applications: nEuro 88 Conference Proceedings*, pages 140–150. IDSET, Paris, 1989.

[25] L.S. Smith. A framework for neural net specification. *IEEE Transactions on Software Engineering*, 18(7):601–612, 1992.

[26] L.S. Smith. Onset–based sound segmentation. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information processing Systems 8*, pages 729–735. MIT Press, 1996.

[27] P.C. Treleaven. PYGMALION: Neural network programming environment. In T. Kohonen, K. Makisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 569–578. North-Holland, 1991.

[28] A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli. *SPICE Version 2G UsersGuide*. University of California, 1981.