ARTICLE IN PRESS

Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science ▮ (▮▮▮▮) ▮▮▮–▮▮▮
Theoretical Computer Science
www.elsevier.com/locate/tcs

Modeling spiking neural networks

Ioannis D. Zaharakis*, Achilles D. Kameas

Research Academic Computer Technology Institute, N. Kazatzaki str., University Campus, GR 26500, Patras, Hellas, Greece

Received 26 July 2006; received in revised form 5 September 2007; accepted 5 November 2007

Communicated by C. Torras

Abstract

A notation for the functional specification of a wide range of neural networks consisting of temporal or non-temporal neurons, is proposed. The notation is primarily a mathematical framework, but it can also be illustrated graphically and can be extended into a language in order to be automated. Its basic building blocks are processing entities, finer grained than neurons, connected by instant links, and as such they form sets of interacting entities resulting in bigger and more sophisticated structures. The hierarchical nature of the notation supports both top-down and bottom-up specification approaches. The use of the notation is evaluated by a detailed example of an integrated tangible agent consisting of sensors, a computational part, and actuators. A process from specification to both software and hardware implementation is proposed.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Formal models; Neural networks; Specification; Systems design methodology

1. Introduction

Broadly an agent is considered as an entity that perceives its environment through sensors and acts upon that environment through its effectors [13,15,7]; all definitions stand to reason that an agent cannot be viewed in isolation from the environment it inhabits. An agent that conforms to the above-mentioned description is generally composed of three types of components: sensor components for perceiving the environment, computational components for deciding the actions (cognitive or communicating) to be performed, and effector components for influencing (directly or indirectly) the environment. Of the components comprising a generic agent, a large amount of the AI research effort (symbolic/logical approaches or subsymbolic/behavioral approaches) is directed towards the computational part.

In order to implement a micro/nanoscale tangible agent, with very limited computational resources, a promising approach is to use spiking neural networks (SNNs). This approach exhibits several advantages and innovations. SNNs are able to simulate certain features of human brain, e.g. decision making or learning, with better accuracy than the classical artificial neural networks (ANNs), especially in applications where time is an important parameter. The innate time aspect of SNNs makes them computationally more powerful than conventional ANNs, implying that they have considerably more processing power than similarly-sized non-spiking neural networks or consequently they

* Corresponding author. Tel.: +30 2610 960200; fax: +30 2610 960490.
E-mail addresses: jzaharak@cti.gr (I.D. Zaharakis), Kameas@cti.gr (A.D. Kameas).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.11.002

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

need fewer nodes to solve the same problem [8,9]. Additionally, SNNs can be implemented directly on VLSI in an extremely compact way providing a number of desirable properties such as self-learning, noise robustness and simple real-world interfaces [6].

One of the problems that the designers face is the formal specification of the network. Because an agent (and thus a SNN embedded into the agent) cannot be studied in isolation from its environment, three separate entities are necessary for the description of the network (introduced by Dorffner et al. in [1], as the "tripartite model"):

1. The *environment component* in which the network is embedded in.
2. The *I/O component* including the pre- and post-processing.
3. The *core component* which is the network itself.

A coarse-grained taxonomy distinguishes ANNs specification approaches (or descriptive frameworks as they are called in neural networks terminology) in languages for describing biologically realistic neurons and networks, object-oriented languages for describing artificial neural networks (including SNNs), mathematical notations for either biologically realistic or artificial networks, and general formal models for specifying parallel systems [12].

Although the languages intended for the description of real neurons do handle time well, they are tightly coupled with the simulation engine and they usually focus on the network itself, thus failing to cover the environment-I/O-core model. Although the object-oriented languages intended for the description of artificial networks allow abstraction and hierarchical description of the objects, they do not handle time well (NSpec [1] is an example exception) and most of them are also coupled with the simulator. Furthermore, frameworks falling in both the above categories are rather language-oriented and thus could be easier classified as simulation tools than typical formal models.

On the other hand, the mathematically-flavored approaches provide powerful theoretical tools describing both structural and functional aspects with formal proofs about the specified network. Moreover, they permit specification at many levels of abstraction, they deal with time-based systems (and thus with temporal neurons) and they easily reflect a tripartite specification framework including spiking neurons, networks, pre-, post-processing steps and the surrounding environment. Finally, the general formal models for parallel systems are useful for hierarchical specification [5,10] and have been influential on the development of formal neural networks models, e.g., [1,11]. Although these models are event driven and specify well the temporal order of events, they encounter some difficulties with the exact timing of events.

Based on the above discussion, the aim of this work is to propose a formal model for the specification of SNNs, which can support various target architectures with emphasis on the direct implementation into hardware (h/w) and its embedding into micro/nanoscale tangible agents. The notation aims to allow either top-down or bottom-up specification and to provide techniques for both the structural description and the execution of a network (in the sense of describing how the structural elements of the model operate and interact with one another over time). It also aims at covering non-neuronal entities participating in the assembly of a tangible agent as well as to deal with the environment in which the network is embedded. Section 2 describes the model itself, illustrating both its structural elements and execution features. Section 3 evaluates the model with the help of an example applying a bottom-up specification approach. Section 4 demonstrates the usage of the top-down specification approach, and Section 5 portrays a development path from formal specification to both software (s/w) and h/w implementation of integrated agents. Finally, Section 6 draws with conclusions and future work.

2. The model

2.1. Influential work

Among the mathematical notations for describing either biologically realistic or artificial networks, two of them partially fulfill the requirements set above. Fiesler in [2] introduces various building blocks of neural network specification and focuses on topological aspects of networks and, as such, this model is useful as a skeleton framework for the classification of the various types of networks. Although the model deals with the transition functions necessary for the dynamics of a network, it does not go into too much detail and some extensions are needed in order to lead to networks recognizable as spiking. However, it does not deal at all with environment and I/O entities.

Smith in [11] proposes a formal model applicable to networks of temporal neurons and by using a sound mathematical notation addresses all the entities of the tripartite approach although no tripartite model is explicitly

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

stated. In order to better reflect the specific characteristics of SNNs and to better deal with time, he uses finer-grained processing elements than neurons (called *nodes*) which interact via instant *links*. All the specified entities (e.g., synapses, activation, output, neuron, network, environment etc.) may be built-up hierarchically allowing either decomposition or refinement and leading to specification detailing or clarity. The main drawback of these models is that they are not executable. As Smith mentions "The notation ... is not (and is not likely to be) executable. It is not a functional specification, but a description of a network (p. 601, [11])".

In order to cope with the above shortcoming, several adaptations and extensions are needed. A convenient approach is to utilize techniques from agent-oriented paradigm that can capture the key functional features of systems like those described in the previous section. For this purpose, a well-suited model is the one proposed by Wooldridge in [14]. Wooldridge's aim was "... an idealized model which captures the most important properties of a wide range of systems (p. 53, [14])". He proposed a theory which includes a formal model for multi-agent systems and a set of execution models defining ways of agents' actions and interactions. Although this formal model was not intended to describe neuron-like computational agents of connectionism, some aspects especially from the executable specification model may be used, properly modified and fused together with those of SNN-oriented models in order to result in a functional formal model for SNNs.

In the following, a formal theory of collaborative autonomous agents comprising of spiking neural networks is presented. The theory consists of a structural model that describes SNNs and of an execution model which shows how groups of nets act and interact with one another. The structural model is inspired from the notion of the "node" as the building block for the construction of SNNs proposed by Smith [11]. Some other structural elements such as the set of constraints, or internal language and epistemic inputs have been proposed by Fiesler [2], Wooldridge [14] and Gardenfords [3], respectively. The execution model handles the above structures and is inspired by the one proposed by Wooldridge [14].

The proposed model does not serve as a theoretical method for investigating the function and mechanism of the nervous system neither it emphasizes the physiology and dynamics of biologically realistic neurons. Instead, it serves as the basis of a neuromorphic engineering approach to provide simulations of spiking neural networks capable of on-chip learning and to build evolvable spiking neural modules directly into hardware.

2.2. Description model

In this section the building blocks (node and link) of a network and their attributes are introduced. Then, all these components are put together in node and net structures.

For the specification of a SNN, it is frequently useful (or in many cases necessary) to describe the structure and the functionality of the participating neurons in detail, e.g. synapses, somas, activation part etc., especially when a hardware implementation is required. This means that there is a need for a building block that can represent any of the parts of a temporal neuron as well as the neuron itself. Thus the notion of the *node* is introduced. Informally, each node contains *ports* and nodes are interconnected with *links*, which are directed arcs that connect nodes via their ports. A set of interconnected nodes comprises a *net*, which exists within an *environment* that is a special case of node.

Let an internal language or symbolism L (e.g. a logical language, a set of alphanumeric characters, the set of real number, pulse type signals, etc.). In order to refer to several structural elements such as nodes, ports and links uniquely, it has been chosen to assign each element an identifier that is node id, port id and link id. The symbols i and pid are used to denote node and port ids respectively whilst the symbol c denotes types of internal language.

$$NodeID, PortID, LinkID = \text{arbitrary countable sets.} \tag{1}$$

The changes in a node's state are represented as *epistemic inputs*. The term "epistemic input" was introduced by Gardenfors to describe "a new piece of evidence" [3]; in our case, epistemic inputs, denoted with ε, are input values to nodes associated with their source. The set of subsets of a type *Form* (i.e. the powerset of *Form*) is given by $\wp Form$.

$$Epin = \wp Form(L) \times PortID. \tag{2}$$

Each node is assigned with a set of constraints expressed in L, which express the node's local state and define the value ranges for the weights, C_W, the local threshold, C_Θ, and the activation values, C_A, (a similar approach has been

taken by Fiesler in his formal definition of a neural network [2])

$$C_W = \wp Form(L)$$
$$C_\Theta = \wp Form(L) \tag{3}$$
$$C_A = \wp Form(L).$$

To avoid confusion with subsequent definitions, we use the term *Situation* to describe a node's local state. A *Situation* is defined to be a triple of the status of the weights, the local thresholds and the activation values.

$$Situation = S_W \times S_\Theta \times S_A \quad \text{where } S_W \in C_W, S_\Theta \in C_\Theta, S_A \in C_A. \tag{4}$$

Nodes contain ports, either input or output. Two constraints apply to ports.

Constraint 1. *An output port is connected to only one input port and vice versa via a directed arc originated from the output port and terminated to input port.*

Constraint 2. *An input port cannot be output port and vice versa.*

$$InPort = \text{the set of input ports} \quad OutPort = \text{the set of output ports} \tag{5}$$
$$Port = InPort \cup OutPort \tag{6}$$
$$InPort \cap OutPort = \varnothing. \tag{7}$$

Now, *Link* is defined to be the set of the directed arcs between the output and input ports that satisfy the Constraints 1 and 2. The symbols l, l' are used to denote links.

$$Link \subset OutPort \times InPort. \tag{8}$$

The auxiliary functions *inport* and *outport* take a link and return the id of the input and output port of the link respectively. The above-mentioned Constraint 2 is expressed with the help of the *inport* and *outport* functions, too (see (10)). Also, Constraint 1 is expressed by (11).

$$inport : Link \rightarrow PortID \quad outport : Link \rightarrow PortID \tag{9}$$
$$\forall l, l' \in Link \cdot ((inport(l) \neq outport(l)) \wedge (inport(l) \neq outport(l'))) \tag{10}$$
$$\nexists l, l' \in Link \cdot ((inport(l) = inport(l')) \vee (outport(l) = outport(l'))). \tag{11}$$

The auxiliary function *nodeport* returns the id of the node that a port belongs to. The auxiliary function *portvalue* takes a port id and returns the value of that port.

$$nodeport : PortID \rightarrow NodeID \tag{12}$$
$$portvalue : PortID \rightarrow Form(L). \tag{13}$$

The combination of (13) with (9) and (10) returns the input and output values of the corresponding ports, i.e.,

$$portvalue(pid) = portvalue(inport(l)) = c \quad \text{where } pid \in PortID, l \in Link, c \in Form(L) \tag{14}$$

returns the value c of the input port *pid*.

Each node contains a function that specifies how the local state of the node is updated and how the outputs are computed (i.e. values placed on output ports). Usually, it is useful to separate the state update part from the port output part. The former is achieved by using a situation revision function (called *SituationRF*), which determines a new set of situations for every situation set and set of epistemic inputs. The latter is achieved by using a transfer function (called *Transfer*), which for every situation set returns the set of epistemic inputs, i.e. the values assigned to the output ports.

$$SituationRF = Situation \times \wp Epin \rightarrow Situation \tag{15}$$
$$Transfer = Situation \rightarrow Epin. \tag{16}$$

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

A *nil situation revision* is assumed, $SituationRF_{nil}$, meaning that a node is not receiving any input or the node is not stimulated, or the node is in its refractory period (the time after a neuron fires during which a stimulus will not evoke a response). A *nil transference* is also assumed, $Transfer_{nil}$, meaning that a node is not producing any output or the node is not activated.

The type for nodes is called *Node*, and it is defined as follows.

Definition 1. A node is a structure:

$$\langle nodeid, Situation_0, port, u, f \rangle$$

where

- $nodeid \in NodeID$ is a node id;
- $Situation_0 \in Situation$ is an initial set of situations;
- $port \subseteq Port$ is a set of input and output ports;
- $u \in SituationRF$ is a situation revision function;
- $f \in Transfer$ is an output revision function.

The environment can be considered as a named node which has input and output ports, an empty situation set and identity situation revision and transfer functions.

A group of named nodes connected with links is called a *net*. The type for nets is called *Net*, it is denoted by the symbol *net* and is given in the following definition. Type *net* may represent a neuron when the nodes represent synapses, activation components etc, or a spiking neural network when the nodes represent neurons.

Definition 2. *A net is a structure:*

$$\langle N, L \rangle$$

where

- $N \subseteq Node$ is a set of nodes;
- $L \subseteq Link$ is a set of links.

It is convenient to define a function which takes a node id and a net and extracts the node associated with the id from the net

$$node : NodeID \times Net \rightarrow Node. \tag{17}$$

Similarly, it is convenient to define a pair of functions which take an input (output) link id and a net and extract the input (output) link associated with the id from the net

$$innet : LinkID \times Net \rightarrow Link \tag{18}$$

$$outnet : LinkID \times Net \rightarrow Link. \tag{19}$$

The proposed model bears two fundamental features inherited by [11]. The nodes exist and evolve over time. Although the name (or id) and the ports of the nodes are invariable, the state, the values of the ports and even the functions do change. The dynamic functionality of the net is expressed by the functions *SituationRF* and *Transfer*. These functions define the behavior of the nodes over time; the whole net behavior over time emerges from the evolution and the interaction of the nodes. The links are identical and they just correspond to instant communication paths; thus they do not represent axonic or dendritic outputs going to many neurons as in biologically neurons or in ANN (see Constraints 1 and 2, or (10) and (11)). This property restricts the characterization of the links as active elements and as a result all the active elements including synapses have to be contained or expressed by the nodes of the net. As mentioned in [11], this approach allows both the subdivision of each neural element (so that

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

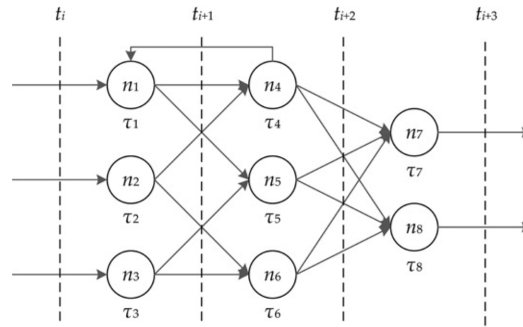

Fig. 1. Synchronous execution model: the circles n_k represent the nodes of the net, τ_k represent internal clock ticks, and $[t_i, t_{i+1}]$ represent single time intervals between two external clock ticks.

nodes may be parts or compartments of a neuron) and clustering of neural elements (so that nodes may be networks of neurons).

2.3. Execution model

The execution model for a net of nodes that is described in the following shows how a group of nodes with the structure described above can operate and interact with one another. Normally, a node receives input via its input ports, updates its state via its situation revision function and eventually fires (according to the evaluation of its transfer function) sending output via its output ports.

In this work, we are interested in the firing of a node as well as for the spike of this firing and not for the internal processes that precede or follow the firing (note that the term "spike" includes all properties of firing, such as potential, intensity, duration etc.). That is why, in the proposed model, we consider node-level synchronous execution assuming that all nodes act in a single time interval. Additionally, the internal node processes (i.e. receiving input, updating internal state and sending output) are considered as indivisible.

The application of a synchronous execution model allows us to phase out the complexity of SNNs that results from their real-time characteristics. These aspects are not necessary, because we only want to study the properties such as liveness and state reachability, which do not depend on real-time aspects of SNN firing but rather on the structure of the network; thus, they can be studied using simulated execution (more details on these properties are presented in Section 5). Thus we do not have to simulate inter-node concurrency; instead, we deal with networks of nodes as a state transition system. To simulate execution time, we use an internal node clock that ticks when the node's internal processes take place and an external clock that ticks when all the internal clocks of the currently enabled nodes have ticked.

An example of the adopted execution model is illustrated in Fig. 1. The circles labeled as n_k, $k = 1, \dots, 8$ represent the nodes of the net. The circle shape for the nodes is preferred in this figure because it hides the currently unnecessary details of the node structure (input ports, output ports etc.). Each node n_k accomplishes its internal processes in τ_k time and this time represents an internal clock tick. A single time interval corresponding to the time between two external clock ticks is represented by $[t_i, t_{i+1}]$. In every single time interval all nodes are executed based on their current state and on the input they receive resulting a new state for the next single time interval.

The execution model relies on the notion of *state* of a net, and of the changes in state being caused by *transitions*. So, a state is an instance of the situation set of each node in the net at some moment in time. A state change, or transition, occurs when one or more nodes receive some input and produce output. The history of an executing net can be considered to be a sequence of states and transitions; a state and a transition are denoted by the symbols σ and τ respectively.

Informally, each node and by extension each net has an initial state. For a node, the initial state is its initial situation set; for a net, the initial state is a collection of initial situation sets, one for each member node. Nodes are able to change the state by performing actions of two types (state update via *SituationRF* and production of epistemic inputs via *Transfer*). A tuple of actions, situation revision and transference, is called a *shift*.

$$Shift = SituationRF \times Transfer. \tag{20}$$

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

The auxiliary function *situation* takes a shift and returns the situation revision of the shift; similarly the auxiliary function *transfer* takes a shift and returns the transference of the shift.

$$situation : Shift \rightarrow SituationRF \tag{21}$$

$$transfer : Shift \rightarrow Transfer. \tag{22}$$

A *nil shift* is any shift which contains the nil situation revision and the nil transference.

The state of a net is defined as a map from node ids to situation sets.

$$State = NodeID \xrightarrow{m} Situation. \tag{23}$$

The initial state of a net, *initial_state*, is defined to be the state where each node has the initial situation set.

$$initial_state : Net \rightarrow State. \tag{24}$$

As more than one node is acting in a net, the actions of each node combine with those of others to change the state of the net. A net *transition* is defined as a map from node ids to the shifts performed by the node.

$$Transition = NodeID \xrightarrow{m} Shift. \tag{25}$$

A *nil transition* is one where each node performs a nil shift. The function *nil_transition* takes a net and returns a nil transition for that net.

$$nil_transition : Net \rightarrow Transition. \tag{26}$$

The operation of a net can thus be described as follows. From the initial state σ_0 of the net, each node makes a shift, which combines with those of others to form a transition, τ_1. From this transition, a new state σ_1 results. The whole process then begins again, with nodes making shifts which form transition τ_2, and so on. The result is a sequence of states.

The auxiliary function *output* returns the set of all the epistemic inputs produced in a transition. The symbol $\triangleq$ can be read "is defined as".

$$\begin{aligned} &output : Transition \rightarrow \wp Epin \\ &output(\tau) \triangleq \{transfer(s) | s \in \mathsf{Range}\, \tau\}. \end{aligned} \tag{27}$$

Similarly, the auxiliary function *input* returns the set of all the epistemic inputs sent to a particular node in a transition.

$$\begin{aligned} &input : NodeID \times Transition \rightarrow \wp Epin \\ &input(i, \tau) \triangleq \\ &\quad \varepsilon = (c, pid) \cdot c \in Form(L), pid \in InPort| \\ &\quad \varepsilon \in output(\tau) \wedge \\ &\quad (nodeport(pid) = i \wedge portvalue(pid) = c). \end{aligned} \tag{28}$$

The function *next_situation* shows how a node's situation changes as a result of executing a cycle on which it performs transference and receives some inputs.

$$\begin{aligned} &next_situation : \\ &\quad Node \times Situation \times \wp Epin \times Transfer \rightarrow Situation. \end{aligned} \tag{29}$$

The function *next_state* for some net returns the successor state under a transition.

$$\begin{aligned} &next_state : Net \times State \times Transition \rightarrow State \\ &next_state(net, \sigma, \tau) \triangleq \\ &\quad i \mapsto next_situation(node(i, net), \sigma(i), \\ &\qquad\qquad input(i, \tau), output(\tau(i)))| \\ &\quad i \in \mathsf{Domain}\, \sigma. \end{aligned} \tag{30}$$

Each transition results a new state, representing the change(s) in the net. Let us call the pair of a transition and the resultant state a *transformation*. Sequences of transformations represent the evolution of the net arising through interaction along its life cycle and extend infinitely into the future. The use of the term "evolution" must not be

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

confused with the evolution through crossover, mutation etc of the evolutionary computation. The symbols t, t', are used for transformations.

The type of transformations is as follows

$$Transformation = State \times Transition. \tag{31}$$

The auxiliary function *state* takes a transformation and returns the state associated with it. The auxiliary function *transition* takes a transformation and returns the transition associated with it.

$$state : Transformation \rightarrow State \tag{32}$$

$$transition : Transformation \rightarrow Transition. \tag{33}$$

The initial transformation of a net is defined to be its initial state together with a nil transition.

$$\begin{aligned} &initial_transformation : Net \rightarrow Transformation \\ &initial_transformation(net) \triangleq \\ &\quad \langle initial_state(net), nil_transition(net) \rangle. \end{aligned} \tag{34}$$

The function *next_transformation* takes three arguments (two transformations and a net) and says whether the second transformation represents a transition of the net from the first transformation. The type $\mathbb{B}$ is assumed to be the set of truth values, i.e., $\mathbb{B} = \{true, false\}$.

$$\begin{aligned} &next_transformation : \\ &\quad Transformation \times Transformation \times Net \rightarrow \mathbb{B} \\ &next_transformation(t, t', net) \triangleq \\ &\quad (state(t') = next_state(net, state(t), transition(t'))). \end{aligned} \tag{35}$$

Transformation sequences are defined to be countable infinite sequences of transformations. If *Evolution* is a sequence of transformations then the *valid_evolution* function validates whether an evolution of a net is valid.

$$Evolution = Transformation^{*} \tag{36}$$

$$\begin{aligned} &valid_evolution : Evolution \times Net \rightarrow \mathbb{B} \\ &valid_evolution(Evl, net) \triangleq \\ &\quad (Evl(0) = initial_transformation(net)) \wedge \\ &\quad \forall n \in \mathbb{N}, n \geq 1 \cdot \\ &\qquad next_transformation(Evl(n-1), Evl(n), net). \end{aligned} \tag{37}$$

The type $\mathbb{N}$ represents the set of natural number, i.e., $\mathbb{N} = \{0, 1, 2, \ldots\}$. Expression $Evl(m)$ denotes the mth element of the sequence *Evolution*.

3. Model evaluation

This section attempts to illustrate an example of how an agent consisting of SNN, sensors and actuators can be specified in terms of the theory described earlier. The specification approach used in this example is clearly oriented to the hardware implementation of the agent per se.

Considering that a specification approach reflects a specific perspective of the system, the specifications given in the following are not definitive but indicative of what is possible. In addition, although the notation used in the following is symbolic, it can be also illustrated graphically. Thus, single line rectangles represent nodes and arrows represent links; an arrow always originates from an output port and terminates to an input port. The application at hand considers an agent which wanders safely in two dimensions. The agent uses three peripherally mounted infrared sensors in order to avoid collisions (sensor components) and two motor wheels (left and right) for its movement (effector components). It can move backwards by reversing the motor wheels and it turns by reducing the speed of the appropriate wheel. The computation component which controls the motor wheels according to the infrared sensors stimuli consists of three integrate-and-fire neurons [4].

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

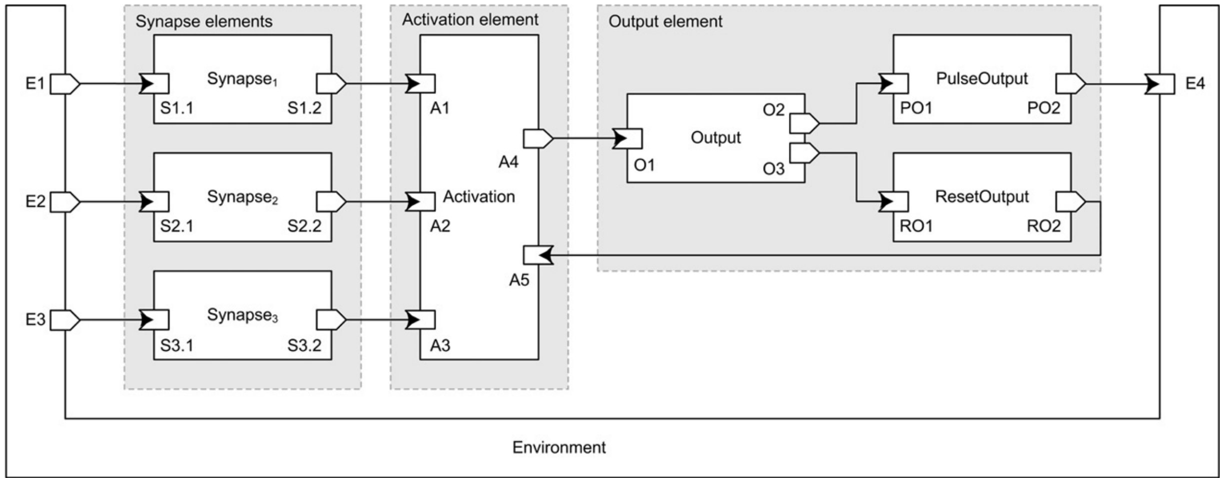

Fig. 2. The integrate-and-fire neuron and its component elements constituted by three synapse nodes, one activation node, and three output nodes responsible for the pulse and the reset signals.

3.1. Neurons description

An integrate-and-fire neuron consists of three component elements: synapse, activation, and output (Fig. 2). The synapse elements are used as follows: one synapse links with a sensor receiving the stimuli readings transformed to pulses and the other two are connected with the two neurons of network via delay distributor nodes, thus representing the delayed synapses of the neuron. All three synapses Synapse_i ($i = 1, 2, 3$) are identical and contain one input port Si.1 and one output port Si.2. The transfer function that describes the behavior of the Synapse_i is the convolution function

$$f_{Synapse_i}(t) = w_i \int_0^t C(x) portvalue(\text{S}i.1)(t - x)\mathrm{d}x \tag{38}$$

where $w_i \in \mathbb{R}$ is the weight of the synapse, $C(x)$ is a convolving function, and $portvalue(\text{S}i.1)$ is the input to the synapse (the description of the integrate-and-fire neuron is based on [12] and [11]). There is no situation revision function for the synapses, the initial activation S_{Ai} is set to 0, and the threshold values $S_{\Theta i} \in \mathbb{R}$.

The activation element contains three input ports (A1, A2, A3) receiving input from the synapses and one fourth (A5) receiving a signal from the output element that will represent the refractory period. The transfer function that describes the behavior of the activation element is represented as a differential equation (for details see [4] and [12])

$$f_{Activation} : \frac{\mathrm{d}A}{\mathrm{d}t} = \sum_{i=1}^{3} \frac{t - T_{\text{RPend}}}{\text{RRP}} portvalue(\text{A}i)(t) - DA. \tag{39}$$

At (39), $T_{\text{RPend}} \leq t \leq (T_{\text{RPend}} + \text{RRP})$, $\text{RRP} \in \mathbb{R}^*$ is the relative refractory period, $D \in \mathbb{R}^+$ is the dissipation subject to $A \geq A_{\text{min}}$, where $A_{\text{min}} \in \mathbb{R}^+$ is the minimum permitted activation. As activation value is part of the node state, the output value of the port A4 is $portvalue(\text{A4}) = A(t)$ when the node fires at time t. As the neuron fires, the Activation node receives a reset signal via A5 input port and the activation value A is reset to 0, thus representing the refractory period. The time when node stops receiving reset signal is recorded as T_{RPend}. The initial activation could be set to 0 (or to any appropriate value) and T_{RPend} such that $T_{\text{RPend}} + \text{RRP} < 0$, so that the neuron is not in its relative refractory period at $t = 0$.

The output element determines when the neuron fires and how long the refractory period lasts. Let, $\varphi \in \mathbb{R}^+$ is the threshold of the neuron, $\text{RP} \in \mathbb{R}^+$ is the refractory period, $L_{\text{max}} \in \mathbb{R}^+$ is the maximum pulse length, and S is the set of possible pulse shapes on the interval $[0, L_{\text{max}}]$. The node Output receives input from the Activation node at time t (via input port O1) and produces output to O2 and O3 if $portvalue(\text{O1}) \geq \varphi$. Then, the node PulseOutput produces a pulse of shape $s \in S$ and the node ResetOutput outputs a reset signal back to the A5 port (let an output value equal to 1) for time RP.

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

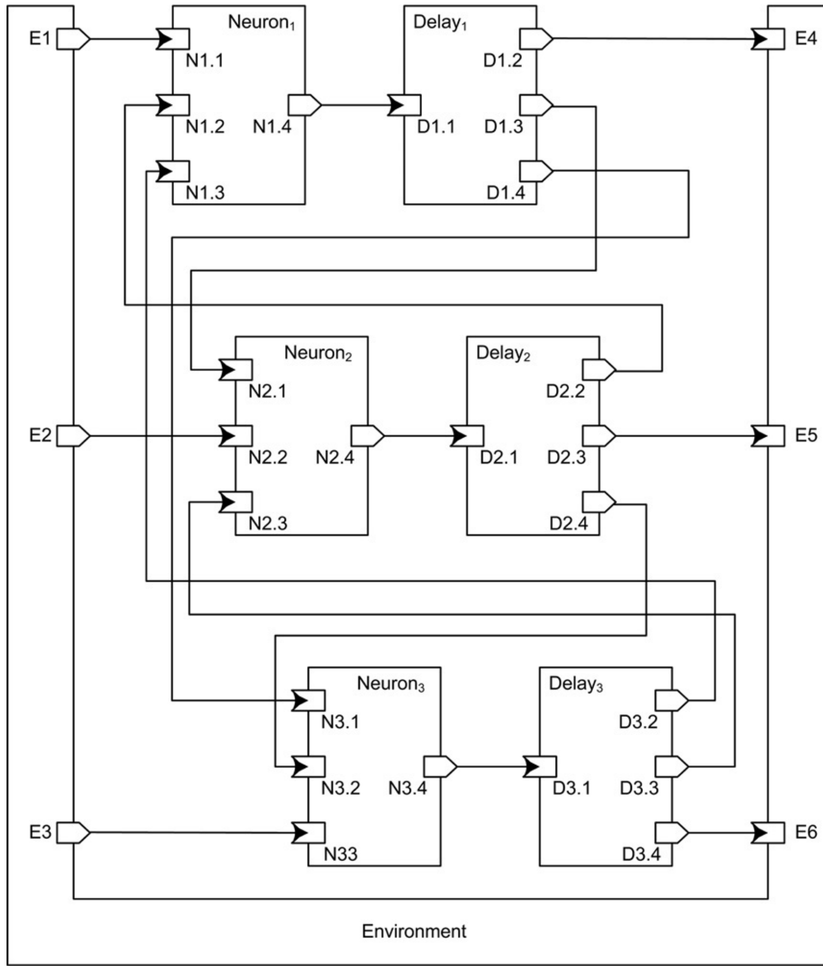

Fig. 3. Network with three integrate-and-fire neurons and delay interconnecting elements. Every Neuron node corresponds to the one above illustrated by Fig. 2. All three Neuron nodes receive input from the environment and the Delay nodes.

The three neurons are interconnected, as illustrated by Fig. 3, via three delay distributor nodes. Each neuron Neuron_i outputs to delay node Delay_i through a link between Ni.4 and Di.1 ports respectively. Then, each delay node Delay_i outputs to the environment as well as to the neurons Neuron_j $(i \neq j)$. This represents the feature where one chosen at random node at a time reevaluates its inputs, producing a new output, or each node reevaluates its output after a random delay time [11]. Considering that each node evaluates its output at the start of each time interval, and that the output is constant, then the transfer function of a delay node can be defined as a vector-value function

$$f_{Delay_i}(t) = \begin{cases} 0 \quad \text{for } 0 \leq t \leq T^i \\ portvalue(Di.1)(t - T^i) \quad \text{for } t > T^i \end{cases} \tag{40}$$

where $\mathbf{T} = (T^1, T^2, T^3)$, $\mathbf{T} \in \mathbb{R}^3$ is the delay vector.

3.2. Sensors and actuators description

Sensors and actuators are the components responsible for the interaction of the agent with the environment. A pair of nodes (SReader_i and PulseGen_i) models the receiving input from an infrared sensor (Fig. 4). The SReader_i nodes transfer the receiving signals to pulse generators and then the signals are transformed to pulses and transmitted to the synapses of the integrate-and-fire neurons. The formal definitions of the transfer functions of these two types of nodes

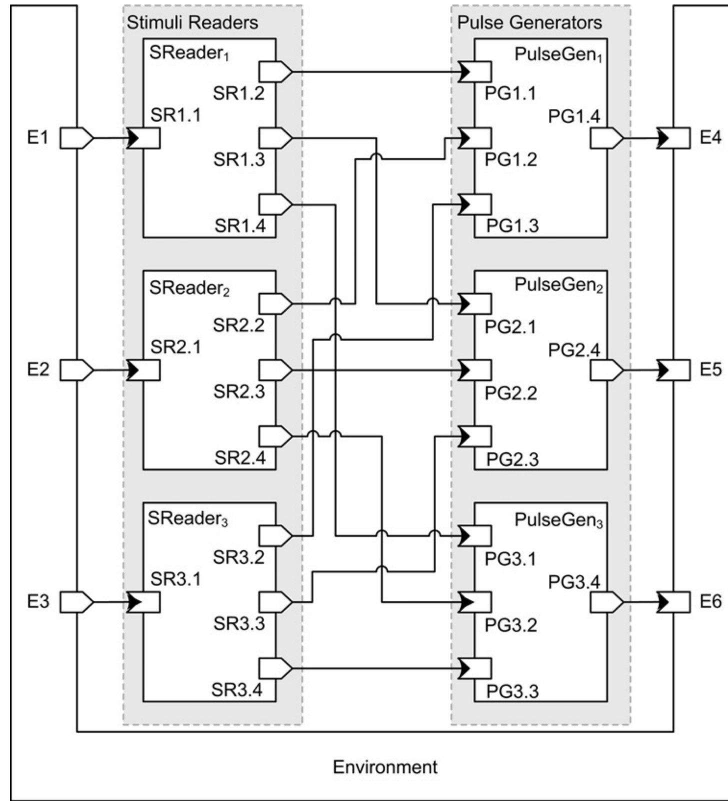

Fig. 4. Nodes of the three infrared sensors: pairs of interconnected stimuli readers and pulse generators receive the input of the sensors through the E1–E3 ports and transmit generated pulses to the integrate-and-fire neurons through the E4–E6 ports. Note that the E1–E6 ports indicate the interface of this particular network with the environment.

are trivial (they just distribute and transform a signal) and are therefore omitted. Input and output ports for SReader_i are SRi.1, and SR$i.j$ respectively, where $i = 1, 2, 3$ and $j = 2, 3, 4$.

On the other hand, the three pulse interpreter nodes distribute the output of the three integrate-and-fire neurons to the action generator nodes (Fig. 5). Then, the ActionGen_i nodes that control the two motor wheels of the robot agent generate the appropriate action. As above, the formal definitions of the PulseInt_i and ActionGen_i transfer functions are trivial and are therefore omitted. Input and output ports for PulseGen_i are PG$i.j$, and PGi.4 respectively, where i, $j = 1, 2, 3$.

3.3. Network description

The construction of the whole network, as illustrated in Fig. 6, is now just the composition of all the subnetworks described above. The environment ports and the corresponding links of the subnetworks are now substituted by the ports and links of the interconnected subnetworks, e.g. environment output ports E1, E2 and E3 in Fig. 3 are now corresponding to the ports PG1.4, PG2.4 and PG3.4, while environment input ports E4, E5 and E6 are corresponding to the PI1.1, PI2.1 and PI3.1 respectively.

Let *ENV* be the environment node. Each neuron is represented by (41) and all the three neurons are represented by (42).

$$Neuron_i = \left(\bigcup_{j=1}^{3} Synapse_j\right) \cup Activation_i \cup Output_i \cup PulseOutput_i \cup ResetOutput_i \tag{41}$$

$$NEURON = \bigcup_{i=1}^{3} Neuron_i . \tag{42}$$

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

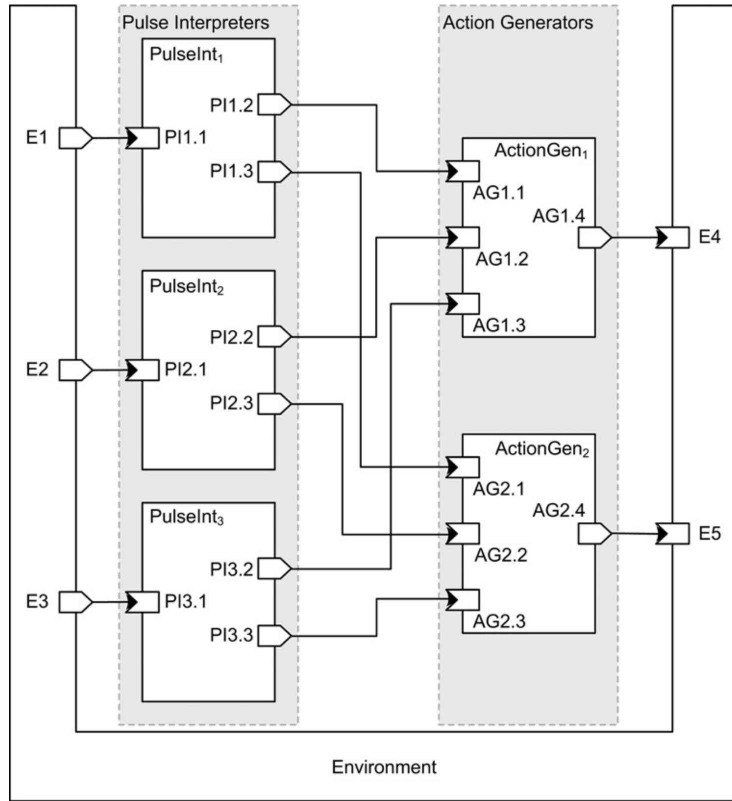

Fig. 5. Nodes controlling the two actuators. E1–E3 ports provide the pulses generated by the three integrate-and-fire neurons. One pulse interpreter node for every incoming pulse feeds two action generator nodes responsible for the control of the wheels. Note that the E1–E5 ports indicate the interface of this particular network with the environment.

Similarly, the sets of delay, sensor and actuator nodes are described by (43), (44) and (45) respectively.

$$DELAY = \bigcup_{i=1}^{3} Delay_i \tag{43}$$

$$SENSOR = \bigcup_{i=1}^{3} (SReader_i \cup PulseGen_i) \tag{44}$$

$$ACTUATOR = \left(\bigcup_{i=1}^{3} PulseInt_i\right) \cup \left(\bigcup_{j=1}^{2} ActionGen_j\right). \tag{45}$$

Then, (46) yields all the nodes of the network.

$$N = SENSOR \cup NEURON \cup DELAY \cup ACTUATOR \cup ENV. \tag{46}$$

The set of links L of the network is produced by applying (9) to all the links subsets that described in Sections 3.1 and 3.2.

Note that the computational component of the above-specified network is based on Smith's description found in [12], although it has been extended mainly to represent different types of output signals. Hence, Smith's description can be used as a reference point and as a subject of useful comparisons. Similarly to Smith's description, the specified network is a recurrent one and this is due to the delayed synapses which are connected with the delay nodes of the other units. Although the particular network is not adaptive, such a feature (if it is desired) can be added straightforwardly. As Smith argues, to maintain locality, information used in altering whatever adapts must be brought to the element in which the adaptation occurs, and this would mean an arc from the output element of a neuron to the synapse elements of that neuron so that Hebbian adaptation could be represented. Nevertheless, depending on the specification

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

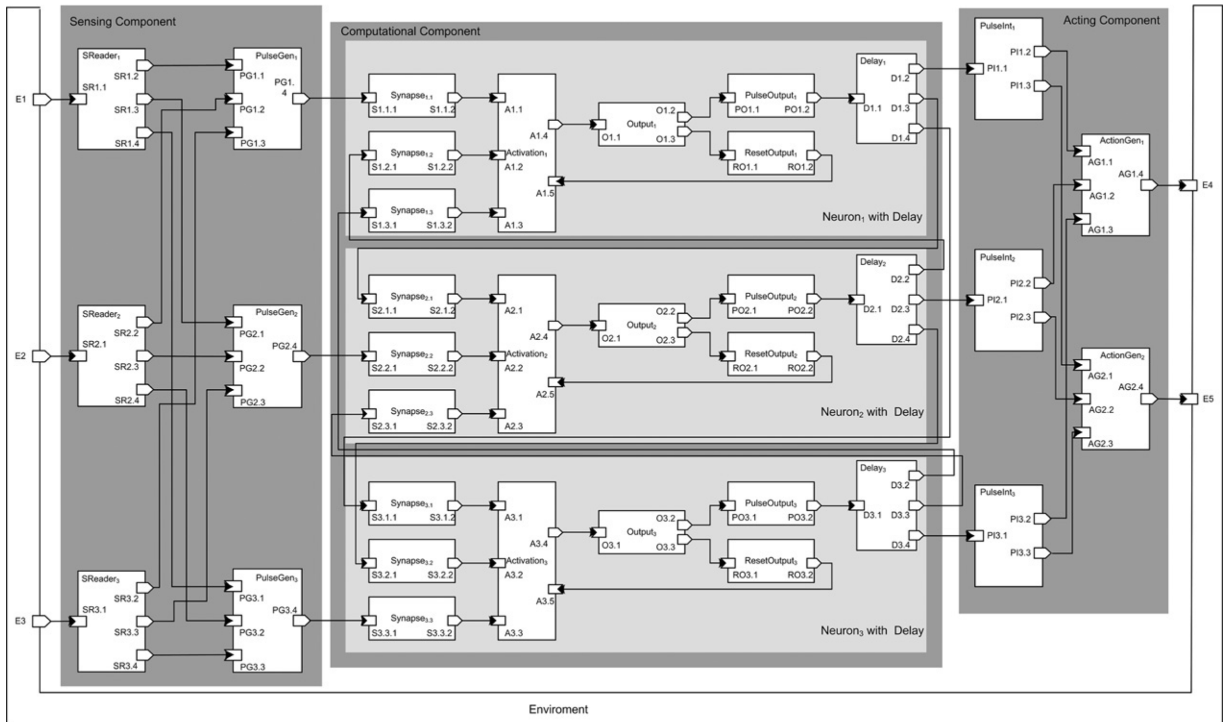

Fig. 6. The specification of the whole network of a robot agent with three integrate-and-fire neurons with delay (Computational Component), three sensors (Sensing Component) and two actuators (Acting Component).

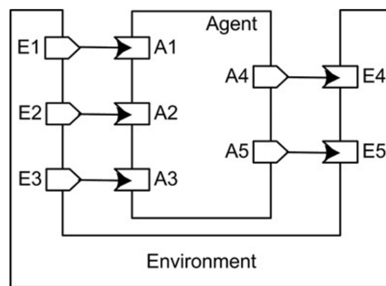

Fig. 7. Top-level diagram: the agent and the interaction with its environment; the A1–A3 ports are responsible for the infrared incoming signals and the A4–A5 ports output to the motor wheels.

perspective and on the target network, a variety of different types of synapse could be used, as for example synapses receiving input from delay elements might be adaptive, and synapses receiving input from the environment may not be adaptive.

4. Top-down specification

This section describes how the proposed model can be used for hierarchical specification. The robot agent example is now gradually decomposed from a high-level perspective and is broken down into manageable pieces.

At the beginning, the agent is placed in its environment and the interaction with the environment is determined (Fig. 7). Essentially, the only parts that are specified are the input and output ports. Clearly, the agent needs input from each of the infrared sensors and produces output to the two motor wheels. The resulting net is

$$\langle Agent \cup Environment, L \rangle \tag{47}$$

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

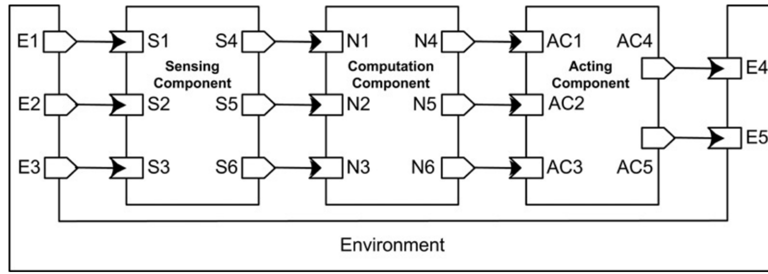

Fig. 8. First-level decomposition: The agent is decomposed to its three basic components.

where,

$$Agent = \langle agent_{id}, s_0, \{A1, A2, A3\} \cup \{A4, A5\}, u_{ag}, f_{ag} \rangle \tag{48}$$

$$Environment = \langle env_{id}, s_0, \{E4, E5\} \cup \{E1, E2, E3\}, u_{env}, f_{env} \rangle \tag{49}$$

$$L = \{(E1, A1), (E2, A2), (E3, A3), (A4, E4), (A5, E5)\}. \tag{50}$$

Next, the Agent node is decomposed into the three basic components that are the sensing, computation and acting ones (Fig. 8). Essentially, both these levels of decomposition reflect the tripartite model of environment-I/O-core components. Notice, that no node functionality or internal state is specified yet, instead only the interconnection between nodes has been determined. Now, (47) is expressed as

$$\langle Sensing \cup Computation \cup Acting \cup Environment, L_1 \rangle \tag{51}$$

where, *Environment* is as (49) and

$$Sensing = \langle sens_{id}, ssens_0, \{S1, S2, S3\} \cup \{S4, S5, S6\}, u_{sens}, f_{sens} \rangle \tag{52}$$

$$Computation = \langle comp_{id}, scomp_0 \{N1, N2, N3\} \cup \{N4, N5mN6\} u_{comp}, f_{comp} \rangle \tag{53}$$

$$Acting = \langle act_{id}, sact_0, \{A1, A2, A3\} \cup \{A4, A5\}, u_{act}, f_{act} \rangle \tag{54}$$

$$L_1 = \{(E1, S1), (E2, S2), (E3, S3), (S4, N1), (S5, N2), (S6, N3), (N4, AC1), (N5, AC2), (N6, AC3), (AC4, E4), (AC5, E5)\}. \tag{55}$$

Advancing to a second-level (or lower-level) decomposition, several decisions have to be taken, e.g. will the network be recurrent, which nodes should participate in a delay process, will the network be adaptive, how the adaptation should occur, which nodes participate in the training, is the network self-organizing, what kind of signals should be transmitted, etc. These kinds of decisions heavily depend on the specification perspective and on the details of the specification process, e.g., to specify the system properties as functional behavior, timing behavior, performance characteristics, or internal structure.

The approach of a second-level decomposition that is followed here is the one illustrated in Fig. 9. The three basic components (sensing, computation and acting) are decomposed into smaller autonomous parts representing the

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

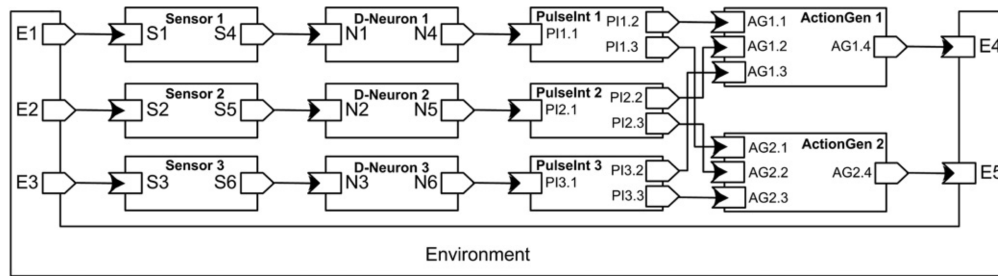

Fig. 9. Second-level decomposition: A specification perspective representing sensors, neurons and (a more detailed subnet of) actuators.

mapping and information flow. A sensor node is assigned to each of the infrared sensors, and its output is passed to a single neuron. As a design convention, the neurons are labeled D-Neuron i, ($i = 1, 2, 3$), where D indicates a delay mechanism participating in a next-level neuron specification. Alternatively, a decomposition of the computational component as the one depicted in Fig. 3 could be adopted. Instead, only the acting component is chosen to be further decomposed and, at this level, is represented by a subnetwork of nodes which receive input from the neurons and combine it in order to generate the action to be performed. The signals transformation functions of the nodes PulseInt i and ActionGen j, ($i = 1, 2, 3$ and $j = 1, 2$) could be specified in this phase.

The formal representation of the second-level decomposition follows similar process as the previous levels and is omitted as trivial. Due to space limitation and to the above-mentioned reasons, no more decomposition levels will be described in this paper. However, it is apparent that someone could easily reach to a level of detail as the one illustrated in Fig. 6 and described by (38)–(46).

5. Integrated approach

From an engineering point of view, the proposed formal model provides the basis for a development path from specification to both s/w and h/w implementation of integrated tangible agents (Fig. 10). As demonstrated in the previous sections, the notation covers the structural and functional specification of either (temporal) neural or non-neural entities (e.g. sensors, actuators etc.). On the other hand, the execution aspect of the model could not only reveal errors and inconsistencies at an early stage but also to be used for the verification of behavioral properties such as liveness and state reachability through simulation of the state transition aspect of the model. These properties are crucial if someone aims at direct implementation into h/w and then embedding into micro/nanoscale tangible agents. This is because one node is implemented into h/w by tens or hundreds of gates (size depends on the node's structure and functions) and a micro/nanosize FPGA has a limited capacity of gates. Thus, it would be not only useful but also necessary, to ensure that all the specified nodes eventually fire in order to avoid non-enabled nodes and deadlocks. Not only the direct h/w implementation but the need for evolution of this module makes the formal model the fundamental representation ingredient standing between the successive generations.

For the above-mentioned reasons, the formal model has been extended into a language and in order to automate the process and to move to fully functional specifications, it is supported by an associated tool entitled CAA Maker (stands for Collaborative Autonomous Agents Maker). CAA Maker is one among a series of tools that support a four-stage CAA life-cycle developed by SOCIAL project funded by EC. These tools are integrated into the SPDE (Social Programmable Development Environment),[1] which main components include (i) CAA Maker responsible for architecture specification and composition modules that generate computational subsystem structures to suit the user specifications, (ii) a simulator for the target physical environments (not described here), and (iii) an evaluator of the genera fitness with respect to the benchmark tasks (not described here, too). Briefly, by using the CAA Maker, the software engineer designs a spiking neural architecture of a module; to accomplish this goal the engineer makes use of the formal model. Software glues in the CAA Maker map these graphs into existing hardware description (e.g. VHDL), which are then directly compiled into evolvable neural hardware. Not only novel semantic transformations are needed

[1] Although a more detailed description of the SPDE would be useful, the current section focuses on the CAA Maker and on how the formal model contributes the development path of integrated tangible agents; so, the interested reader is referred to related material of the SOCIAL project.

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

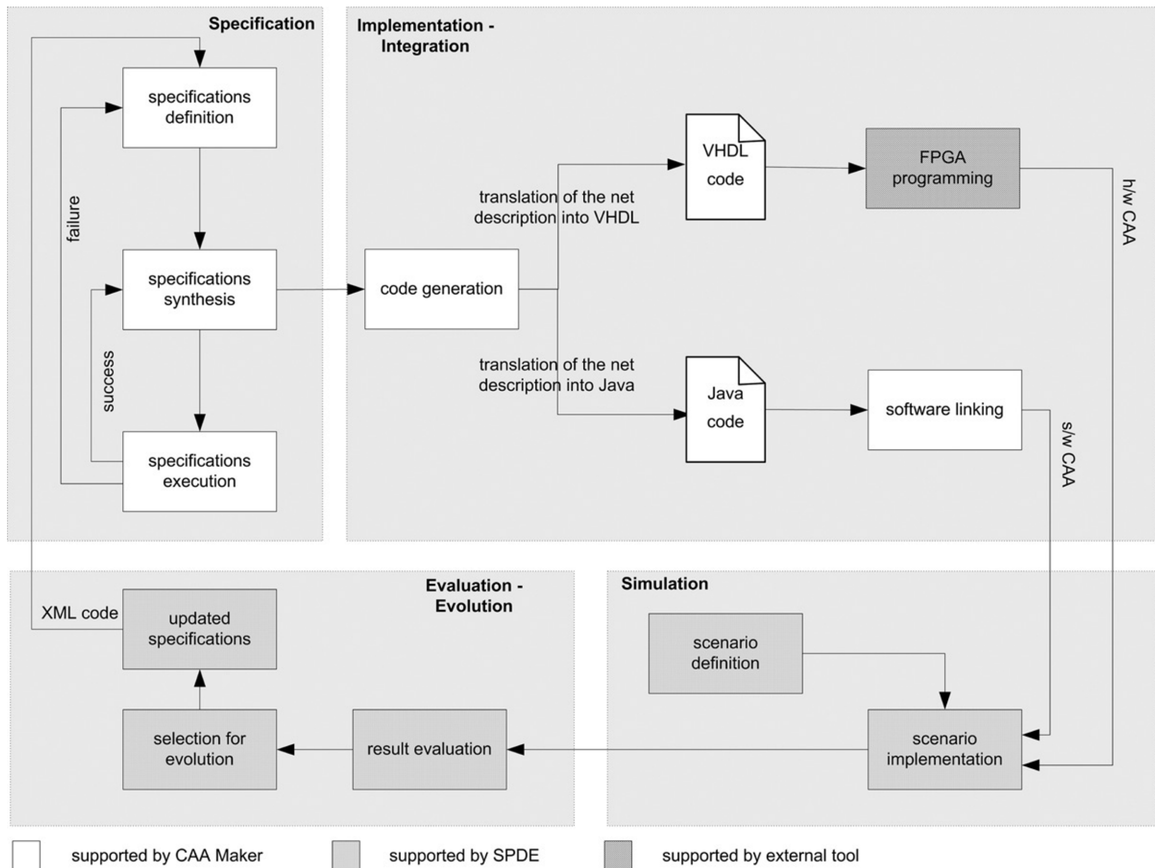

Fig. 10. The path from specification to both s/w and h/w implementation of integrated tangible agents.

to automap topologies onto hardware, but a new way of thinking (aptly described as "software paradigms in hardware design"). A team of identical agents that harbour the aforementioned architecture is formed and then tested in the simulator using software models of sensors and actuators and of the target physical environment. Sensory impressions and actuator commands to and from the neural hardware are communicated using existing hardware/software communication protocols. The individuals are allowed to learn as they explore the dynamic environment and after an initial adaptation period, genera fitness is evaluated according to preset mission specifications. Subsequently the set of rules that generated the hardware architecture is modified accordingly by an evolutionary (genetic) algorithm. Then, the evolved system is fed back to the CAA Maker in order to produce the new generation into s/w and h/w.

CAA Maker provides a uniform method to deal with specification, implementation and verification processes of the several components comprising a tangible agent. Not only does it hide the strict specification formalism by reflecting it into graphical structures but also provides a pure object-oriented language to program the specified components; additional, it translates into different target languages. Also, CAA Maker provides compatibility with W3C compatible tools as it stores and interprets files in XML format. To achieve the multiple code generation a cross-compiler translates the graphical representation into:

- XML code, which describes the structural characteristics of the net-based formalism as well as the functionality of the participating nodes.
- VHDL code, which can be used for the programming of FPGA modules.
- Java jar files, which, along with the XML description can be used by a simulator in order to test the behavior of the CAAs.

The design and implementation of the CAA Maker is based on and consistently reflects the formal specification model. Designing an SNN, or in general a CAA, can be seen as two different phases. In the first one, simple nodes (the basic building blocks of CAA Maker) are combined to create complex nodes, which may represent neurons, synapses,

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

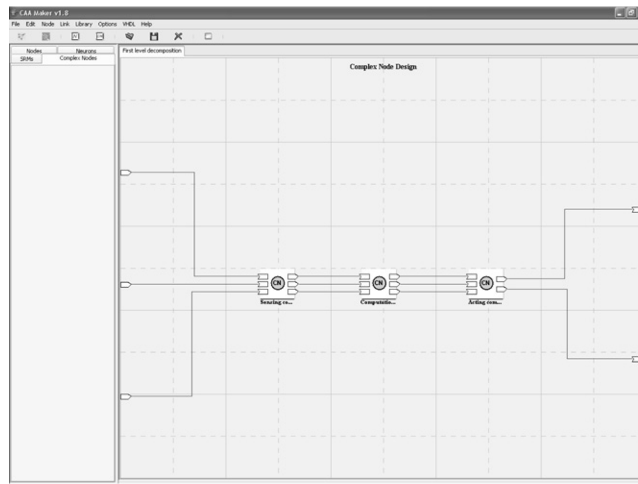

Fig. 11. The first-level decomposition as designed by CAA Maker.

activations, or other types of design elements. This process aims to the structural design of the net. In the second phase, the functionality of the previously created structures is defined. This is achieved by programming the nodes with the employed CNL language (a Java-like subset language). Every intermediate design can be stored/exported to XML for reuse by the designer or for use by other components of the SPDE.

Briefly, the CAA life-cycle model consists of the stages of Specification, Implementation–Integration, Simulation, and Evaluation–Evolution (Fig. 10). The *Specification* stage focuses on the specification of the structure and dynamics of SNN (computational component) as well as the sensor/actuator components. Based on the formal model, a designer is able to specify a spiking neural architecture for the computational subsystem. Furthermore, as the stage deals with the overall architectural design other necessary parts of the CAA can also be described (e.g., glue components, basic behavior components, etc.) and used as library components. The *Implementation–Integration* stage includes the compilation of the specifications and the production of XML descriptive code, Java-like subset language for node function definition and VHDL code. The Java-like subset language are used for the implementation of the s/w CAA, while the VHDL code is used for the implementation of the h/w CAA. The specifications are stored in XML descriptive code based on W3C standards; the XML descriptive code describes the structure and functionality of the CAAs as they are stored in library components. The *Simulation* stage deals with the XML code and the Java-like subset language and how they can be used by simulation tools in order to produce executable s/w CAAs and test them into a simulated environment. In parallel, the h/w CAA can also be tested by the simulation tools so as to verify the execution of SNNs in hardware. All the simulations are performed according to certain scenario settings in order to evaluate the performance of the two products. The *Evaluation–Evolution* stage evaluates the CAAs according to measurement results and success criteria and produces evolved CAA in XML code which comprises input to the Specification stage and a new cycle begins.

CAA Maker supports the first two stages. The software engineer specifies the networks both structurally and functionally, which are internally represented (stored/exported for later use) by XML schemes (“specifications definition” process). These networks are compiled together (“specifications synthesis” process) and can be verified through simulation (“specifications execution” process). Successful designs (in terms of verified specifications) constitute the input of the code generator (“code generation” process), where VHDL, Java and XML codes are produced. The VHDL code can be used (though needs some kind of optimization) for direct implementation into FPGA boards, the Java code forms the s/w counterpart, whilst the XML code constitutes a standardized intermediate representation compatible with many software tools (e.g. simulation packages). In CAA Maker and in an attempt to facilitate the specification process, the mathematical net structure defined by Definition 2 is referred as “complex node”; as such, complex nodes are only abstraction structures and their functionality results by the specified functionality of the nested nodes (Definition 1). Fig. 11 illustrates the net of Fig. 8 as it is designed by using CAA Maker; its XML code is represented by the Fig. 12. Figs. 13 and 14 depict the structure of the whole net and of the “Computational component” node, respectively, produced in VHDL code.

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

```
- <COMPLEX_NODE>
   + <USER_INFORMATION>
     <COMPLEX_NODE_NAME>First level decomposition</COMPLEX_NODE_NAME>
     ... graphics information
   - <NODES>
      + <COMPLEX_NODE>
      - <COMPLEX_NODE>
           <COMPLEX_NODE_NAME>Computation component</COMPLEX_NODE_NAME>
           ... graphics, decomposition information
           <INPUTS_NUMBER>3</INPUTS_NUMBER>
         - <INPUTS>
            - <INPUT>
                 <INPORT_ID />
                 <INPORT_WEIGHT>0.0</INPORT_WEIGHT>
                 <INPORT_NAME>N1</INPORT_NAME>
              </INPUT>
              ... N2 and N3 inport definition
           </INPUTS>
           <OUTPUTS_NUMBER>3</OUTPUTS_NUMBER>
         - <OUTPUTS>
            - <OUTPUT>
                 <OUTPORT_ID />
                 <OUTPORT_NAME>N4</OUTPORT_NAME>
              </OUTPUT>
              ... N5 and N6 outport definition
           </OUTPUTS>
           <COMPLEX_NODE_ID>ComplexNode@3</COMPLEX_NODE_ID>
           <C_LOCATION>477;417</C_LOCATION>
         </COMPLEX_NODE>
      + <COMPLEX_NODE>
     </NODES>
   - <LINKS>
      - <LINK>
           <LINK_ID>Link@3</LINK_ID>
         - <INPORT>
              <COMPLEX_NODE_ID>ComplexNode@3</COMPLEX_NODE_ID>
              <INPORT_ID>ComplexNode@3.0</INPORT_ID>
              <INPORT_WEIGHT>1.0</INPORT_WEIGHT>
            </INPORT>
         - <OUTPORT>
              <COMPLEX_NODE_ID>ComplexNode@1</COMPLEX_NODE_ID>
              <OUTPORT_ID>ComplexNode@1.0</OUTPORT_ID>
             </OUTPORT>
        </LINK>
         ... rest of the links definition
     </LINKS>
     <INPUTS_NUMBER>3</INPUTS_NUMBER>
   - <INPUTS>
     ... definition of the inputs from the environment node
     </INPUTS>
     <OUTPUTS_NUMBER>2</OUTPUTS_NUMBER>
   - <OUTPUTS>
     ... definition of the outputs to the environment node
     </OUTPUTS>
</COMPLEX_NODE>
```

Fig. 12. Net representation in XML code: this code describes the network structure illustrated by Fig. 11.

6. Conclusion

A formal specification model covering both the structural and dynamic aspects of spiking neural networks has been proposed. The proposed model integrates a structural SNN modeling approach [11,2] with agent-based execution

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY e_First_level_decomposition IS
PORT (
E1 : IN STD_LOGIC; E2 : IN STD_LOGIC; E3 : IN STD_LOGIC;
E4 : OUT STD_LOGIC; E5 : OUT STD_LOGIC
);
END ENTITY;

ARCHITECTURE a_First_level_decomposition of e_First_level_decomposition IS

COMPONENT Sensing_component
PORT(
S1 : IN STD_LOGIC;  S2 : IN STD_LOGIC; S3 : IN STD_LOGIC;
S4 : OUT STD_LOGIC; S5 : OUT STD_LOGIC; S6 : OUT STD_LOGIC
);
END COMPONENT;
COMPONENT Computation_component
PORT(
N1 : IN STD_LOGIC; N2 : IN STD_LOGIC; N3 : IN STD_LOGIC;
N4 : OUT STD_LOGIC; N5 : OUT STD_LOGIC; N6 : OUT STD_LOGIC
);
END COMPONENT;
COMPONENT Acting_component
PORT(
AC1 : IN STD_LOGIC; AC2 : IN STD_LOGIC; AC3 : IN STD_LOGIC;
AC4 : OUT STD_LOGIC; AC5 : OUT STD_LOGIC
);
END COMPONENT;

BEGIN
Sensing_component1 :Sensing_component
PORT MAP(
S1 => E1, S2 => E2, S3 => E3,
S4 => N1, S5 => N2, S6 => N3
);
Computation_component2 :Computation_component
PORT MAP(
N1 => S4, N2 => S5, N3 => S6,
N4 => AC1, N5 => AC2, N6 => AC3
);
Acting_component3 :Acting_component
PORT MAP(
AC1 => N4, AC2 => N5, AC3 => N6,
AC4 => E4, AC5 => E5
);
END a_First_level_decomposition;
```

Fig. 13. File “First-level decomposition.vhd”: VHDL code generated by CAA Maker for the first-level decomposition of the agent.

semantics [14]. The basic building block of the model is the node, a finer-grained structure than the neuron. Thus, the model describes both networks and neurons as sets of interacting entities. The provided abstractions permit specification at higher level and support top-down and bottom-up approaches as well. From a dynamic point of view, the model reasons about the operations and the interactions of the network entities and provides a means for the verification of certain behavioral properties. To achieve this, the model uses node-level simulated execution and hides the internal calculation processes of each neuron. Thus, the model can deal with several types of networks as well as several types of neurons (temporal or non-temporal); it can even be used to describe non-neural entities, e.g. sensing

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY e_Computation_component IS
PORT (
Computation_component_INPORT0 : IN STD_LOGIC;
Computation_component_INPORT1 : IN STD_LOGIC;
Computation_component_INPORT2 : IN STD_LOGIC;
Computation_component_INPORT0 : IN STD_LOGIC;
Computation_component_INPORT1 : IN STD_LOGIC;
Computation_component_INPORT2 : IN STD_LOGIC
);
END ENTITY;

ARCHITECTURE a_Computation_component of e_Computation_component IS

BEGIN

END a_Computation_component;END a_First_level_decomposition;
```

Fig. 14. File “Computation component.vhd”: VHDL code generated by CAA Maker for the complex node “Computational component” of the agent.

and acting components. Finally, the notation, being platform independent and not tied to any specific tools, simulation packages, etc., sets a path to implementation on both software and hardware architectures.

Acknowledgements

This work was supported in part by a grant from the European Community under the “Information Society Technologies” Programme (01/01/2003-31/12/2005), Project SOCIAL IST-2001-38911 (http://www.socialspike.net).

References

[1] G. Dorffner, H. Wiklicky, E. Prem, Formal neural network specification and its implications on standardization, Computer Standards and Interfaces 20 (4) (1999) 333–347.
[2] E. Fiesler, Neural network classification and formalization, Computer Standards and Interfaces 16 (3) (1994) 231–239.
[3] P. Gardenfors, Knowledge in Flux, MIT Press, 1988.
[4] W. Gerstner, Time structure of the activity in neural network models, Physical Review E 51 (1) (1995) 738–758.
[5] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985.
[6] T. Lehmann, R. Woodburn, Biologically-inspired on-chip learning in pulsed neural networks, Analog Integrated Circuits and Signal Processing 18 (2) (1999) 117–131.
[7] M. Luck, P. McBurney, C. Preist, Agent Technology: Enabling Next Generation Computing – A Roadmap for Agent Based Computing, AgentLink II, 2003, also available at http://www.agentlink.org, 2004.
[8] W. Maass, Networks of spiking neurons: the third generation of neural network models, Neural Networks 10 (9) (1997) 1659–1671.
[9] W. Maass, Computation with spiking neurons, in: M.A. Arbib (Ed.), The Handbook of Brain Theory and Neural Networks, 2nd edition, MIT Press, 2002, pp. 1080–1083.
[10] R. Milner, Communication and Concurrency, Prentice Hall International, 1989.
[11] L.S. Smith, A Framework for Neural Net Specification, IEEE Trans. Software Eng. 2 (7) (1992) 601–612.
[12] L.S. Smith, Using a framework to specify a network of temporal neurons, in: Proc. First Slovak Neural Networks Symposium, 1996, pp. 111–127.
[13] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd edition, Prentice Hall, 2003.
[14] M. Wooldridge, The logical modelling of computational multi-agent systems, Ph.D. Dissertation, Dept. of Computation, Univ. of Manchester UK, 1992.
[15] M. Wooldridge, in: G. Weiss (Ed.), Intelligent Agents, Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, MIT Press, 1999, pp. 27–77.

Please cite this article in press as: I.D. Zaharakis, A.D. Kameas, Modeling spiking neural networks, Theoretical Computer Science (2007), doi:10.1016/j.tcs.2007.11.002