SPEC *perceptron*
internal CHAN $Dendrite_i, Error_i$, **external** CHAN $Training_i$
internal REAL o, t
internal CONSTANT N=number-of-inputs, M=number-of-neurons
$\langle\langle \| : 0 < j \leq N, 0 < i \leq M :: synapse_{ij} \rangle$
$\| \langle\| : 0 < i \leq M :: \langle$**while true**
$\langle\langle$**ready** $\rightarrow Dendrite_i$? $o \|$ **ready** $\rightarrow Training_i$? $t \rangle \, [] \, Error_i$! $(o - t) \rangle\rangle\rangle$

4. Conclusions

All neural languages try to introduce a tool for high level specifications with maximal flexibility which covers a wide set of physical processing architectures. Unfortunately such a flexibility often conflicts the final goal of the specification task. The proposed alternative is a formal language that allows the specification of neural networks to be given as a low level architecture. Communication between the units and synchronization issues are explicit. The given formalism does not rule out specifications of abstract neural models; because most of the models have a biological background and a biological system is an architecture there should not be any problems in specifying any type of artificial neural systems.

REFERENCES

[1] Rumelhart D.E. & McClelland, J.L (Eds.) (1986) *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Vol 1., Cambridge, MA: MIT press.

[2] D'Autrechy, C.D., Reggia, J.A., Shutton III, G.G. & Goodall, S.,M. (1988 July), A general purpose simulation environment for developing connectionist models. *Simulation,* pp.5-19.

[3] Hecht-Nielsen H. (1990) *Neurocomputing*.Reading, MA: Addison-Wesley Publishing Company.

[4] Goddard N.H, Lynne, K.J., Mintz T., Rochester Connectionist Simulator, Tech.l Rep. 233, Univ. of Rochester, CS Dept., March 1988.

[5] Paik E., Gungner, D. and Skrzypek, J. (1987). UCLA SFINX-A neural network simulation environment. *Proceedings of the IEEE 1st International Conference on Neural Networks*, pp.III-367-III-375.

[6] Hanson, W.A., Crutz, C.A., Tam, J.Y. (1987). CONE -Computational Network Environment, *Proceedings of the IEEE 1st International Conference on Neural Networks*, pp. III-531-III-545.

[7] Ernst H.P., Morky, B. & Schreter Z., (1990). A Transputer based general simulator for connectionist models. *Proceedings of the Parallel Processing in Neural Systems and Computers*, pp. 283-286. Amsterdam: Elsevier Science Publishers B.V.

[8] Gumm, H. P. & Hergert, B. H. (1990). Neural Pascal: a language for neural network programming. *Proceedings of the Parallel Processing in Neural Systems and Computers*, pp. 287-290. Amsterdam: Elsevier Science Publishers B.V.

[9] Lester, B. L. (June 1987). Neural network simulation language based on multi-Pascal. *Proceedings of 1st IEEE International Conference on Neural Networks,* pp. III-347-III-354.

[10] Heilman, Brown and Georgiopoulos (1990). Simulation of artificial neural network models using an object-oriented software paradigm. *Proceedings of the IEEE International Conference on Neural Networks*, pp. II-133-II-136.

[11] Kraft, T., Frostrom, S. A., MacRitchie, B. & Rogers A. E. (1989). *The Specification of a Concurrent Backpropagation Network using Actors.* San Diego, USA: Science Applications International Corporation.

[12] Bavan A.S. (1989). NIL-PLUS: a neural network implementation language. *Proc. First Neural Computing Meeting*, London, pp.171-178, The Institute of Physics.

[13] Hoare, C.A.R. (1978). Communicating Sequential Processes. *Communications of the ACM, 21*(8), 666-676.

[14] Koikkalainen P. and Oja E. (1988), Specification and implementation environment for neural networks using communicating sequential processes. *Proceedings of the IEEE International Conference on Neural Networks*, pp. I-533-I-540. San Diego, CA.

the process definition: *channel-variable* **?** *variable*.

3.4 Process synchronization

In MIND there is no external synchronization between parallel processes but synchronization is still possible via guarded communication processes. Guarded communication allows the user to operate MIND channels either synchronously or asynchronously. This freedom of choice is introduced via the channel control flag which the corresponding sending or the corresponding receiving process sees as a boolean variable **ready** which is **true** if the corresponding flag is zero and **false** otherwise. This variable can be used as a guard that may given in the *channel-process* definition. The value of the *guard* may also be a result of any boolean expression and the corresponding process is stopped until the value of the guard is true. If channel guards are used with the *conditional-process* then a guard may point to the corresponding channel directly as shown in the following example. The MIND process is equivalent to the Occam (for Occam syntax see [14]) process.

$\langle((a+b<10) \wedge \textbf{ready}) \rightarrow chan_1$? x
$\|$ $\langle$**if**
(**ready** $chan_2$) $\rightarrow \langle chan_2$? y $[]$ $proc_2 \rangle$
$\|$ (**ready** $chan_3$) $\rightarrow \langle chan_3$? z $[]$ $proc_3 \rangle\rangle\rangle$

PAR
$(a+b<10)$ & $chan_1$? x
ALT
$chan_2$? y
$proc_2$
$chan_3$? z
$proc_3$

The structure of the channel allows the user to mix synchronized and unsynchronized channel processes with the same channel variable. As a result any type of deterministic or nondeterministic process control mechanism can be realized.

3.5 Variable bindings

In MIND new variables may be defined for any process. A declaration binds a variable in the scope of the following process. Parallel processes do not share variables, only communication mechanism between parallel processes is via a *channel-variable* which is a special case. Sequential processes and processes down in hierarchy may share variables. For example in ⟨**int** x,y process-a $[]\langle$process-b $\|$ **int** z process-c $\rangle\rangle$variables "x" and "y" are in the scope of all three processes but "process-b" or "process-c" may have its own copies of these variables and a change of value for "x" in "process-b" is not seen in "process-c". The rule is that only one of the parallel processes may change the value of a variable that in the scope of all of them and the others have their own copies of the variable. A change of "x" or "y" in "process-a" is seen both in "process-b" and in "process-c". Variable "z" is only in the scope of "process-c".

3.6 Example specification: Perceptron

SPEC $synapse_{ij}$
external CHAN $Input_j, Error_i, Dendrite_i$
internal REAL $x, err, weight$
internal CONSTANT $\alpha = 0.05$
$\langle$**while true** $\langle\langle$**ready** $\rightarrow Input_j$? x $[]$ **ready** $\rightarrow (+)Dendrite_i$! $x * weight \rangle$
$\|$ $\langle$**ready** $\rightarrow Error_i$? err $[]$ $weight := weight + \alpha * err * x \rangle\rangle\rangle$

if $a < b$ then arbitraryly the $process_1$ or the $process_2$ is executed.

The implementation of a *conditional-process* may be a list of sequential and parallel guarded processes which share a state variable. The state variable is initially **false** and the guards of these processes are executed on the order of the list until one of the guards returns **true** in which case a process is executed and the value of the state variable is put **true**. If all the guards of the list return **false** then the *conditional-process* behaves like the **stop** process until the guard of the last sequential process or one guard in the last set of parallel processes becomes **true** and the corresponding process is executed.

An *assignment-process* specifies evaluation of an expression and assignment of the result to a variable. It also specifies input and output to and from the *channel-variable* which is shared between parallel processes. The first form of the *assignment-process* allows communication between sequential processes and the second form makes it possible for parallel processes.

$$assignment\text{-}process \longrightarrow variable := expression \mid channel\text{-}assignment$$

An assignment with := assigns the value of expression on the right of := to the corresponding variable on the left. This is similar to assingments in any language like PASCAL and C. A *channel-assignment* is similar to the channel assignment in CSP with some extended features.

$$\begin{aligned} channel\text{-}assignment \longrightarrow\ & channel\text{-}variable\ \mathbf{!}\ channel\text{-}expression \\ \mid\ & channel\text{-}variable\ \mathbf{?}\ variable \\ \mid\ & guard \rightarrow channel\text{-}variable\ \mathbf{!}\ channel\text{-}expression \\ \mid\ & guard \rightarrow channel\text{-}variable\ \mathbf{?}\ variable \end{aligned}$$

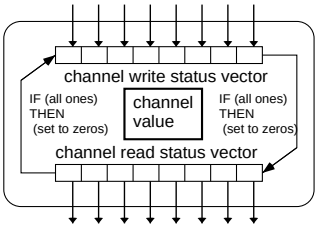

Figure 1. The channel architecture

3.3 Channels

A channel can be realized by using a *channel-variable* which is shared between parallel processes. Two vectors of boolean flags are also required to control the set of sending and the set of receiving processes: there is one flag for every process. This also implies that the number of processes communicating via a channel must be known before the actual use of the channel. Initially all flags controlling the sending processes are set to zero and all flags controlling the receiving processes are set to one. If the values of all sending flags or all receiving flags are ones then the corresponding control vector is said to be in wait mode. During an operation a process may set or read the value of the *channel-variable* and after any read/write operation the corresponding flag is set to one. When there is a change of mode in the control vector, last zero flag is set to one, all flags in the other vector are set to zero and the *channel-variable* is set to zero. This allows several ways of process synchronization to be used with a channel, the architecture of which is depicted in figure 1.

Processes that are sending are defined by using "**!**" type of assignment and the receiving processes are defined by using "**?**". The sending process assigns the value of the *channel-variable* to the value of the *channel-expression* on the left of "**!**". A *channel-expression* an extended expression, the value of which is defined by

$$channel\text{-}expression \longrightarrow\ +(\ expression\ ) \mid expression$$

In the case of $+(\ expression\ )$ the value is the value of the corresponding *channel-variable* plus the value of the *expression*. This form is useful when a process is a model of a dendritic tree of a neuron. The receiving process assigns the value of the *channel-variable* to a *variable* given in

The *variable-declaration* is similar to any other languages:

$$\textit{variable-declaration} \longrightarrow \textit{type variable-name } \{,\textit{variable-name}\}$$

Allowed variable types are not explicitly specified; all normal types are allowed. The *channel-declaration* defines a channel variable that two or more parallel processes can use for communication. MIND channels can receive inputs from many processes and they may send a value to many processes in parallel. Communication mechanism is extented to nondeterministic behavior: both the receiving and the sending process can use the channel without any synchronization mechanism. A channel is defined by

$$\textit{channel-declaration} \longrightarrow \text{ CHAN } \textit{channel-identifier } \{,\textit{channel-identifier }\}$$

3.2 Processes

The basic primitive is a process. A process may be one of the basic computing structures or a combined structure defined by the *process-declaration*. There are three special processes: **stop**, a process that starts but never terminates; **skip**, a process that terminates when it starts and **wait**, a process that terminates after *time* of seconds from it's start. None of the special processes have any side effects.

$$\begin{aligned}\textit{process} \longrightarrow\ & \textit{parallel-process} \mid \textit{sequential-process} \mid \textit{conditional-process} \\ & \mid \textit{assignment-process} \mid \textbf{stop} \mid \textbf{skip} \mid \textbf{wait } \textit{time}\end{aligned}$$

A *parallel-process*, defined by using "$\|$", is a set of processes that are executed in parallel

$$\textit{parallel-process} \longrightarrow \{\| \textit{ process }\} \mid \langle \| : \textit{ replicator} :: \textit{ replicate-process } \rangle$$

The *replicator* is any bound variable and it's scope. The *replicate-process* is a set of processes where every occurrence of the bound variable (or variables) is replaced by a numeral defined by the scope. For example, $\langle \| :$ **(int)** $i = [1 \ldots 3]$ :: my-process$_i$ $\rangle$ stands for

$$\text{my-process}_1 \parallel \text{my-process}_2 \parallel \text{my-process}_3$$

Thus the bound variable i is replaced for every occurrence in "my-process" with the given index. A *sequential-process* defines an ordered set of processes that are executed sequentially

$$\textit{sequential-process} \longrightarrow \{[] \textit{ process }\} \mid \langle []: \textit{ replicator } :: \textit{ replicate-process } \rangle$$

A *conditional-process* defines a guarded execution of sub-processes. The *conditional-process* is combination of an alternative command [13] and an **if-then-else** type of structure. The execution of the *conditional-process* will result an execution of exactly one sub-process defined by guards that are boolean expressions. The notation is

$$\begin{aligned}\textit{conditional-process} \longrightarrow & \textbf{if } \textit{guard} \rightarrow \textit{process } \{[] \textit{ guard} \rightarrow \textit{process} \mid \| \textit{ guard} \rightarrow \textit{process}\} \\ & \mid \langle \textbf{if} : \textit{ replicator} :: [] \textit{ replicate-guard} \rightarrow \textit{replicate-process } \rangle \\ & \mid \langle \textbf{if} : \textit{ replicator} :: \| \textit{ replicate-guard} \rightarrow \textit{replicate-process } \rangle\end{aligned}$$

The executed sub-process is selected either as the first *process* of which the boolean guard is true from an ordered set of processes or as an arbitrary process of which the boolean guard is true from an unordered set of processes. In case of ordered processes the guarded commands are separated by using delimiter [] and in the case of unordered processes the delimiter is $\|$. For example in $\langle$**if** $(a < b) \rightarrow process_1 \parallel$ **true** $\rightarrow process_2 \rangle$the $process_2$ is executed if $a \not< b$ but

network terms and the model architecture is rather general. In the Rochester Connectionist Simulator [4] and in the UCLA SFINX [5] the network structure is described by using the C programming language routines, which are linked to the simulation environment directly. Other neural specific extensions to the C language are proposed for example in [6] and [7].

2.2. Model independent languages for ANS specifications

Object oriented languages are attractive for model independent specifications of ANS, for a recent discussion see [10]. The abstract nature of object oriented programming may also be an obstruction that is not easy to overcome. It is alluring to use high level objects in a way that makes it very hard, if not impossible, to compile the specifications onto hardware architectures. One of the best known specification languages which is based on objects is ANSpec [11]. NIL-PLUS (a Neural Network Implementation Language) [12] is closer to the architectural specification of ANS than most languages. There are two languages in NIL-PLUS: a network specification language and a manipulation language. Manipulation language allows a user to specify simulation related information such as initial values and feeding of the training data; the network architecture itself is defined in a network specification sub-language. Model independent neural languages which are based on Pascal are introduced in references [8] and [9].

3. MIND: a specification language for artificial neural networks

In this section a new formalism called MIND (Massively Interconnected NonDeterministic processes) is introduced. There are two goals that MIND specifications have to fulfill: it must be a simple theory that applies to wide range of applications and it must be specific to neural computing. An existing theory that fits quite nicely to many of these requirements is CSP [13]. The connection between neural networks and CSP was first introduced in reference [14]. In this work the CSP theory is extented to fit better to neural network specifications.

MIND is a modular formalism for specification purposes; not a true programming language. The reader should be familiar with some basic structuring methods such as replicators, conditionals and assignments. The programming notation is introduced using BNF. Curly braces $\{\}$ indicate that a unit in the braces may be instantiated zero or more times.

3.1 The program structure

A MIND program is divided into sections. The first section is used for declarations and the second section is the functional description of the program itself. A MIND program is a process that may communicate with other processes. The process itself can be a set of parallel running processes. The *process* may be one of the basic structures or it may be defined by a *process-declaration*:

process-declaration$\longrightarrow$ **SPEC** *process-name*
{**external** *declaration-list*}
{**internal** *declaration-list*}
$\langle$ *process* $\rangle$

Local variables and local channels are defined after key word **internal**. The *process-name* is a string, process name, that binds to a list of commands defined by the *process*. Operands are defined by the *declaration-list*:

declaration-list$\longrightarrow$ $\{$*channel-declaration* $\}|\{$*variable-declaration*$\}|\{$*process-declaration* $\}$

MIND: A Specification Formalism for Neural Networks

Pasi Koikkalainen

Lappeenranta University of Technology
Finland

A specification formalism for artificial neural systems (ANS) is introduced. In contrast to many other specification languages the given formalism will suggest unique and straightforward implementations of ANS. The proposed concept is an extension of "Communicating Sequential Processes" (CSP) with neural network related features. A discussion of other "neural" specification languages is also given.

1. Introduction

The objective is to specify complex neural systems via a formalism or a language that is natural for the system that is being developed. Specification process must take into consideration several aspects like easy design, implementation efficiency, clarity and flexibility. Also, specification should not needlessly increase the implementation complexity or restrict the system characteristics.

The modeling of neural networks should separate concerns between the specification and the implementation. From computational point of view neural networks are not programs, they are an architecture. The specification must describe the architecture while the implementation may be a computer program or a hardware realization of that architecture. High level descriptions of neural networks which specify only a program are useful for simulation experiments: they are easy to built and easy to understand. Any simulation experiment, however, is not practical without a model architecture which allows the description of the neural network to be given in a structured environment. Without such a framework there is no advantage in using general purpose "neural" programming languages in place of a programming language of any kind.

2. Specification languages for neural systems

Many "neural" languages are based on traditional, sequential, languages and used with a model of neural networks that binds the language to a simulation environment. Such a framework is useful when implementing paradigms on a specific type of system but it also restricts the portability and may even be harmful when doing innovative development of new neurocomputing paradigms. Other alternative is to use languages that are independent of models but then the difficulty is to find a formalism that is natural for the specification task.

2.1 Languages based on a model of ANS

In reference [1] ANS simulator P3 is described. P3 uses LISP as a specification language under restrictive programming framework. Use of LISP as a network specification language was also reported in [2]. AXON [3] is a language that combines features from object oriented programming and from several traditional languages. The language syntax is based on neural

This work was supported by Tekes Grant 4196/1988 under the Finsoft project.