A LOTOS-Based Approach To Neural Network Specification

J. Paul Gibson
Department of Computing Science,
University of Stirling,
Stirling FK9 4LA.
(jpg@uk.ac.stir.cs)

11th May 1993

Abstract

The objective of the SPLICE[1] project is to help bridge the gap that currently exists between customers, specifiers and designers. Central to the initial phases of the project is the identification of problem domains suitable for study, together with the prototyping of tools for the specification and animation of systems within the chosen problem areas. Neural network specification has been identified as one area which can be usefully considered. This paper examines how LOTOS can be used to specify the behaviour of one particular type of neural network: a trained perceptron. It also gives an overview of the *customer-friendly* environment, currently under development, which provides tools for the generation and testing of perceptron models, defined as LOTOS specifications. We conclude with recommendations for further research into the area of neural network specification using LOTOS.

[1]SPLICE: Specification using LOTOS for an Interactive Customer Environment.

1 Introduction

1.1 Introducing SPLICE

The initial formal statement of requirements is a difficult, yet vital part of system development. Errors should be identified and removed as early on in the development process as is possible. Identifying a higher proportion of errors in the initial development stages depends on improving the communication between customers and designers. The customer must be able to comprehend the formal requirements model in their own terms. Comprehension is aided when the static structure of the problem domain is represented in the structure of the requirements specification, together with the identification of some correspondence between components of the problem domain and components of the FDT being used to model the requirements. Further, interactive execution of the formal model can improve customer understanding of the dynamic properties of the system being specified.

The work in SPLICE (the initial project proposal, which gives a good overview of the objectives of the work, is given in [TUR]) first identified a number of problem domains suitable for further analysis. The domain of neural computing was chosen, as a suitable domain, for the following reasons:

- There has, quite recently, been much interest expressed in the formal specification of neural networks.
- Neural computation provides a large, diverse and expanding audience for FDTs, in general, and LOTOS, in particular.
- Neural networks are highly concurrent, dynamic systems composed from sets of fundamentally very similar components, with well defined notions of structure.

1.2 Introducing Neural Networks

1.2.1 Parallel Distributed Processing Systems (PDPSs)

PDPSs [RUM] occur in many guises in computing science. Any system (hardware or software) which is composed of independently executable communicating processes can be said to be a PDPS. The key feature of a PDPS is the distribution of processing and data. This makes the interpretation of such data more difficult than with non-distributed systems. However, it is such a flexible representation method which gives rise to many of the perceived advantages of distributed data, e.g. gradual degeneration of system behaviour due to data corruption or hardware failure.

1.2.2 Neural Models

In neural computing there are a large number of models — each a type of PDPS. The processes, and interprocess communications, in neural networks are modelled on the behaviour of neurons in the brain [BEA]. These models are called neural networks [KOH]. A general characteristic of neural networks is the simplicity of behaviour at each node process, so that the complex behaviour of a network arises from the large number of interactions between nodes rather than from complexity inherent in the node behaviour.

There are many algebraic formalisms which are suitable for the representation of PDPSs, [HOA, MIL] are two well known examples. Furthermore, there have been various attempts, based on these formalisms, towards a characterisation of a general model for neural network specification and design, e.g *MIND* in [KOI] and *AXON* in [HEC]. Unfortunately, there is no standard way of specifying an artificial network; or of prototyping, testing and implementing such a specification. However, many of the different models used in the specification of neural networks share the same features. The first step in providing a generalised framework for the development of neural network specifications is the identification of these common features.

1.2.3 A Framework for the Specification of Neural Networks

Neural network models currently fall into two catagories: informal and executable, or formal but non-executable. The advantage of an executable model, with respect to improving customer understanding, is that the dynamic behaviour of the network can be explored, through animation, in a step-wise fashion. Informal models are prone to ambiguity and are not amenable to mathematical manipulation. Formal mathematical models, such as proposed in [SMI], can be used to specify neural network behaviour but cannot be directly simulated for prototyping or testing. What is required is a formal description technique which provides facility for direct execution of specifications, whilst allowing specification at different levels of abstraction.

The description of a neural network needs to define the following:

- The static structure of the interconnectivity between processes.

- The interface between the network and its environment (static and dynamic aspects).
- The dynamic behaviour of the system over time: the type of communication between processes and the internal processing performed by each node.

Any model which describes these three aspects can be said to provide a specification of a neural network.

1.2.4 The Advantages of Formality

The need for formality in the description of neural networks has been identified [ZIP, SMI]. As neural networks increase in complexity, and the science of neural computing becomes more widely followed, a concise and unambiguous method of describing networks increases in importance.

At the simplest level, a standard formal specification technique allows researchers to include specifications in papers without risk of misinterpretation. Reasoning about the behaviour of neural models can then be done in a clear and consistent fashion. A mathematical framework would prove its usefulness through the increase in mutual understanding between neural scientists.

Neural network behaviour, as described by a formal language, can be mathematically analysed without the need for execution. The type of static analysis that can be usefully employed is currently under investigation. In particular, we are examining the role of non-standard interpretations (views) of the LOTOS code. It is hoped that this information can be used to improve customer understanding and be incorporated in a tool for automatic documentation generation.

A FDT is vital when considering the re-use of component specifications. There are many risks in re-using library components if their behaviour is not formally defined. Although formality is not necessary for re-use (consider most programming languages), it is necessary for controlled re-use.

2 Modelling Neural Networks Using LOTOS

2.1 Modelling a Framework of Inter-process Communication

LOTOS is well suited to the modelling of networks of static and dynamic intercommunicating processes [FRE]. The static topology of a neural network can be realised directly in LOTOS — each neuron in the network has a corresponding process in the LOTOS specification. The communication between processes can be modelled synchronously or asynchronously. A LOTOS specification facilitates a formal static analysis of certain features of a neural network. Animation of the LOTOS specification allows neural network designers to preview the behaviour of the network before an implementation is coded in an efficient manner. Mathematical manipulation of the specification, in the form of formally defined transformations, may also benefit neural network developers (although, our research has yet to verify this claim).

2.2 Architectural Concepts and Structured Development

The basic components of a neural network can be realised as 'LOTOS templates'. Node processes, communication links, node layers, node slabs, input/output slabs, fan-out units, fascicles, etc., [HEC] are specified as process algebra components. Learning functions, connectivity functions, connection signal types, neural state, etc., [HEC] are specified using the ACT ONE part of LOTOS.

LOTOS encourages a structured development of specifications/designs. Composition of a neural network is recognised by a LOTOS specification of a network of inter-communicating nodes being described as a single node. Decomposition involves a node process being replaced by a different neural network (sub)system. Different parts of a neural network can therefore be viewed at different levels of abstraction.

2.3 Re-Use: An Object Oriented Flavour

The current research on neural network specification has identified the need for node generators [HEC], i.e. library components or templates for the construction of parameterised node behaviour. [ZIP] acknowledges the suitability of object orieted techniques in this area. Node behaviours can be classified in a hierarchical fashion and inheritance provides one type of re-use mechanism. There is reason to believe that (sub)network topologies and learning mechanisms could also be classified for re-use. A standard set of neural network components would free designers from having to repeatedly express the same architectural knowledge in different network models. LOTOS, being amenable to an object oriented style of specification [RUD, HUL], may provide a suitable formal basis for this type of behaviour re-use. This is an area of current research in SPLICE.

2.4 Potential Problems for Specification

In many neural networks, the environment is a time varying stochastic function over the space of input values [RUM]. In other words, at any time t, there is some probability that any of the input values is impinging on the input processes/nodes. The specification of general neural systems requires a model that can cope with synchronous, assynchronous and continuous time systems. Continuous time systems are notoriously difficult to model in an executable environment. There may also be difficulties in modelling the global time necessary for synchronous dynamics if structured top-down development is to be allowed (i.e. the decomposition of one network process into a sub-network of component processes may not be possible within the constraints of a global time model).

Although there is a large body of research with regard to the specification of timing ([BOL] is a good example of such work in the LOTOS domain) and probabilistic ([HAN] gives an overview in a general process algebra framework) properties in different process algebras, it was felt unwise to try and incorporate this (not yet fully mature) work in our initial investigation of neural network specification. Rather, we chose to examine a type of network which does not rely on these types of semantics (and so can be specified using LOTOS in a standard way): a perceptron.

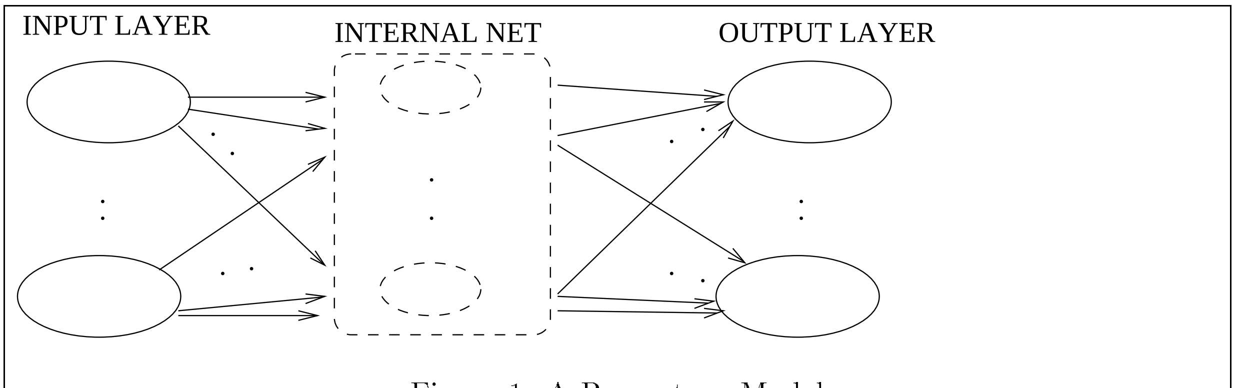

Figure 1: A Perceptron Model

3 The Neural Network Model: A Perceptron

3.1 Perceptron Structure

Figure 1 illustrates the structure common to all perceptrons. Our LOTOS model permits any form of `Internal Net`, even though the set of all possible perceptron models is equivalent, in behaviour, to the set of perceptrons with only one hidden layer [RUM]. Each of the nodes in the perceptron model has the same structure, as illustrated in figure 2. The node fires the value `weightOut`, on all its output lines, when the sum of its inputs ($\sum_{i=0}^{n} W_i x_i$, where $x_i \in \{0,1\}$) is greater than (or equal to) the threshold value intrinsic to the node.

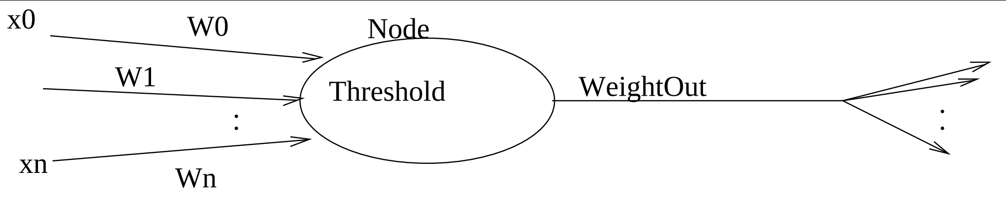

Figure 2: A Perceptron Node

3.2 Learning: An Overview

The perceptron learns by adjusting the weights on the links between nodes. These adjustments are made as follows:

- When the output is active, but it should be inactive, then the internally active units have their `WeightOut` values reduced. In our model this reduction is calculated to be the sum of the signals being received by the node.
- When the output is passive, but it should be active, then the internally active units have their `WeightOut` values increased. In our model, the increase is again the sum of the input signals.

These adjustments model the correction of misbehaviour. The environment of the network is required to judge the correctness of the output, for each set of inputs, and prompt the weight changes. In our LOTOS model the changing of weights is done through the `update` gate.

3.3 The Generation of Perceptron Networks: Two Examples

The LOTOS specification of the perceptron model is given in appendix A. To test this specification we generated a number of different networks. Two examples of the networks which were generated and tested are: a simple XOR gate, and a noughts and crosses pattern recogniser. We consider these two cases before the specification is considered in more detail.

3.3.1 An XOR Perceptron

The XOR problem is often used as a standard test for neural models. The network which we generated to solve the problem of learning XOR behaviour has structure as illustrated in figure 3.

Fortunately, in this simple case, we managed to execute the specification so that it learned how to distinguish between two sets of inputs: $\{00, 11\}$ and $\{01, 10\}$.

3.3.2 A Noughts-and-Crosses Perceptron

To test our perceptron model more completely, we defined a more difficult problem: recognising *three-in-a-row* in the noughts and crosses game. In this case, there are nine input nodes (one for each position on the board). An active input corresponds to an X. An inactive input is not considered. Three Xs in a row should produce an active output. Conversely, an inactive output

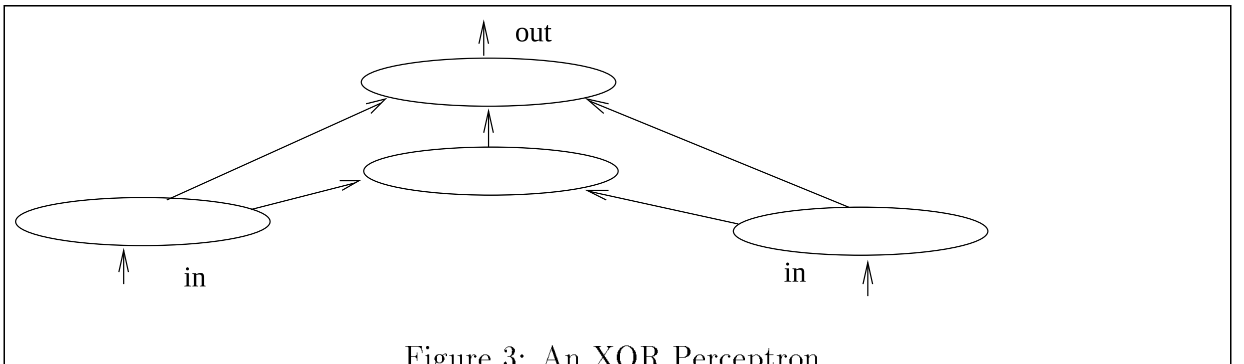

Figure 3: An XOR Perceptron

occurs when there are not three Xs in a row. The perceptron designed to solve this problem was structured as illustrated in figure 4.

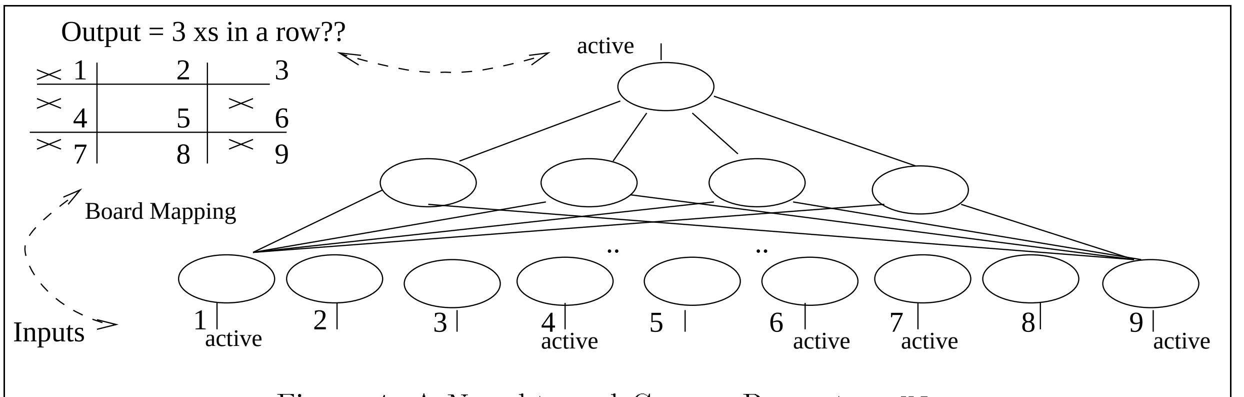

Figure 4: A Noughts and Crosses Perceptron:`XOs`

Unfortunately, although we were able to generate such a neural network, it was not possible to train it by a simple animation process (using LITE). After approximately a dozen learning iterations, the network did not appear to have learned very much. In this case, it was necessary to implement the model to achieve a performance level necessary for learning. The `XOs` perceptron was useful only to clarify how the network was going to learn. The specification was too unwieldy to model learning during an animation of *reasonable* duration.

3.3.3 Lessons Learned From Examples

The primary lessons learned were:

- It *is* possible to usefully specify neural network behaviour in LOTOS.
- The specifications are not *customer oriented* in their present form, but there is potential for formalising the standard graphical representations, using LOTOS.
- The animation tools need extension before they can be usefully applied in the execution of a learning process which takes many iterations.

4 The LOTOS Perceptron Specification: An Overview

The code for the Perceptron network is given in the `NETWORK` specification in appendix A. This is complete, except that the ACT ONE equation definitions are not included. Rather than examining this code in detail, we give an overview of its structure, and extract some of the more interesting specification fragments.

4.1 The `NetSpec` Process

The `NetSpec` is defined to have two distinct stages of behaviour. In the first stage, the network of nodes and links is constructed. The structure is defined in the ACT ONE sorts which parameterise the `NNet` and `InOut` component processes: `Net` is used to represent the internal structure of the network and `InOut` represents the external interface, i.e. the set of `In` and `Out` nodes. When construction is complete, a `complete` event occurs. This partial behaviour is defined in the specification fragment below:

behaviour
NetSpec[inn,out,linkin,linkout,addlink,addnode,complete,update] (NoNodes,NoNodes)
where process NetSpec[inn,out,linkin,linkout,addlink,addnode,complete,update] (InSet:NodeIDSet,OutSet:NodeIDSet):**noexit**:=
hide message **in**
(NNET [addlink,addnode,complete,message,update](N_N(N0),None)
| [linkin,linkout,complete,addlink,message] |
InOut[inn,out,linkin,linkout,addlink,complete,message](INSet, OutSet))

The left hand side of figure 5 shows the structure of the system before the `complete` event occurs. After completion the structure in the ACT ONE is transferred to the process algebra: all internal nodes are defined as `NNode` process instances. This transformation is done by the `MakeNet` process:

```
process MakeNet[message,update](NS1:NetState): noexit:=
hide madestep in
([IsEmpty(NS1)] -> stop) []
([not(IsEmpty(NS1))] ->
( NNode[message,update]
(getID(getNode(NS1)), getFr(getNode(NS1)), getTo(getNode(NS1)),
getTh(getNode(NS1)), getWe(getNode(NS1)))
| [message, update] |
(madestep; MakeNet[message,update](remove(NS1)))))
```

Once all the added nodes have been realised as `NNode` processes, the `MakeNet` process (see the right hand side of figure 5) stops and the behaviour of `MakeNet` is defined by the set of `NNode` processes running in parallel. These communicate with each other using the `message` gate and synchronise with the environment on the `update` gate. The `ModelIO` process controls the learning process through the gates `in` and `out`. It guarantees that all inputs are received before the output is calculated:

```
process ModelIO[inn,out,message]
(InS:NodeIDSet,OS:NodeIDSet):noexit:=
(GetIns[inn,message](InS,OS,InS) >>
(GiveOuts[message,out](InS,OS,OS) >>
ModelIO[inn,out,message](InS,OS)))
```

4.2 The `NNode` Process

The `NNode` process defines how each node communicates with the others, and how it learns from its environment. The `NodeInt` process synchronises with all `message` events (internal node to node communications), but passes on (using the `MessageOn` gate) only those communications which the node directly participates in. The `Node` component process rotates between: getting all inputs from its input lines, sending its resulting output on its output lines and updating the weighting on its output lines (in response to an `update` event):

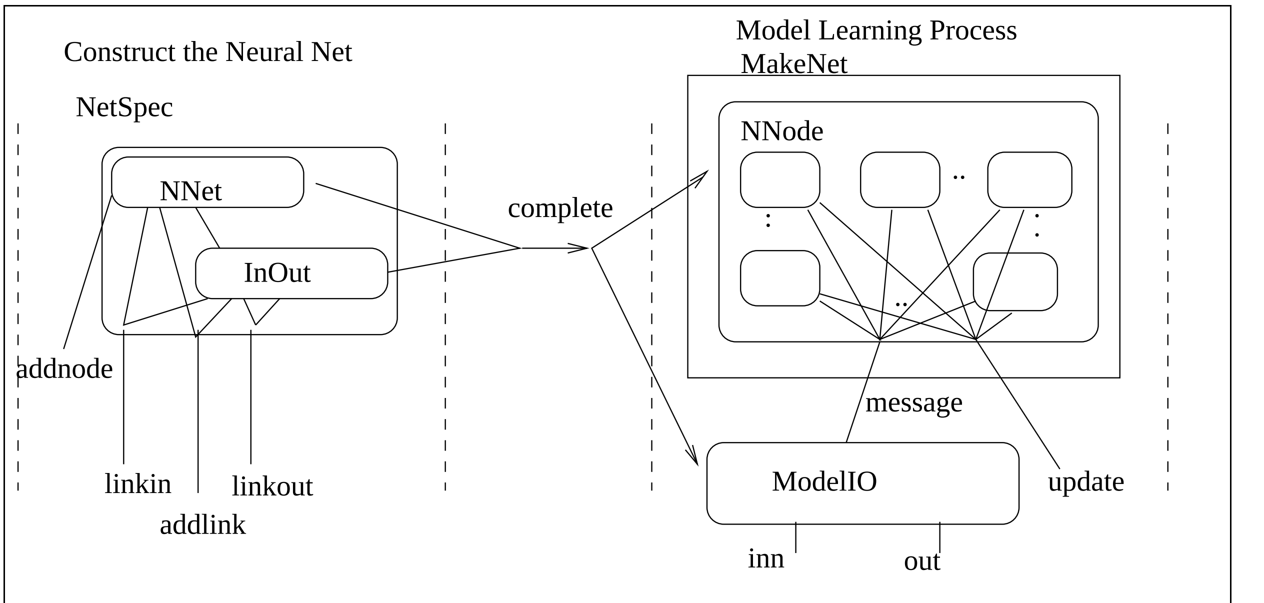

Figure 5: A High Level View of the Perceptron Specification

process NNode[message,update]
(ID:Node,CommFrom:NodeIDSet,CommTo:NodeIDSet,
Threshold:Num,WeightOut:Num):**noexit**:=
hide MessageOn **in**
(NodeInt[message, MessageOn](ID,CommFrom,CommTo)
| [MessageOn] |
Node[MessageOn,update](ID,CommFrom,CommTo,Threshold,WeightOut))

4.3 Notes On Specification Style and Clarity

The specification is not written in any particular style. Work is currently being done in an attempt to specify the perceptron model in an object oriented style, but this is not yet complete. The specification is not easy to understand from a simple browsing of the code. However, most LOTOS users gain a high degree of their understanding through interaction with a specification using the standard tools. Our initial goal was to make the specification *customer accessible*. At this stage in the work it is clear that the standard LOTOS tools are not appropriate for neural network designer interaction. There is a definite need, in this instance, to provide problem domain specific tools. In particular we need to transform the LOTOS models onto standard ways of representing perceptron systems. This work is currently in progress.

5 The Development Environment

A useful neural network development environment must incorporate one of the following:

- Some means of automating specification generation and manipulation.
- A static analysis tool and associated document generation facility.
- A tool for automating the dynamic behaviour of the specification.

When formally modelling neural networks, the relatively simple core behaviour of the network nodes must be embedded in a system that lets the customer (i.e. the neural network developer) observe, interact with and manipulate their functionality. Currently, a suitable user-interface is being developed on an X-Windows platform.

6 Further Work

As this paper reviews only a preliminary investigation in a larger body of work, it is easy to identify many more aspects of the problem which need further consideration. The following are currently under investigation, or in the process of development:

- Automation of the process of generating the neural networks.
- Prototyping of the development environment.
- Extending the work to other types of neural networks, in particular those which can learn without guidance.
- Testing of the LOTOS specifications against neural network designer's requirements.
- Relating the work on neural networks to other problem domains in an attempt to identify general principles and practices.

7 Conclusions

Neural network specification is fundamentally concerned with capturing and testing the functional and structural requirements of parallel distributed processing systems. The perceptron example, given in this paper, shows that LOTOS is well-suited to this task.

This paper reports on an initial investigation into the production of formal specifications with *customer bias*. Even at this early stage, we believe that it is both possible and desirable to build and test LOTOS requirements models which are amenable to customer interaction, albeit with comprehensive tool support.

This paper shows that neural network specification using LOTOS is worthy of further investigation.

Acknowledgements

The author acknowledges the contributions of his fellow SPLICE workers, namely Ashley McClenaghan and Ken Turner. Also, Bob Clark, Peter Ladkin, Ana Moreira and Dominique Mery must be thanked for their input.

References

- [**BEA**], *Neural Computing: an introduction*, R. Beale and T. Jackson, Adam Hilger IOP Pub. 1990.
- **BOL**], *LOTOS-like process algebras with urgent or timed interactions*, T. Bolognesi and F. Lucidi, in: 'Formal Description Techniques IV', pp 249 — 264, (ed. K. Parker, G. Rose), North-Holland 1992.
- [**FRE**], *Modelling Dynamic Communicating Structures in LOTOS*, L. Fredlund and F. Orava, in: 'Formal Description Techniques IV' (ed. K. Parker and G. Rose), 1992.
- [**HAN**], *A framework for reasoning about time and reliability*, Hans Hansson and Bergt Jonsson, in: Proceeedinggs 10th IEEE — Real Time System Symp., Santa Monica, pp 102 — 111, Computer Society Press, 1989.
- [**HEC**], *NeuroComputing*, R. Hecht-Nielson, Addison-Wesley, 1990.

- **[HOA]**, *Communicating Sequential Processes*, C.A.R Hoare, Prentice-Hall International, 1985.
- **[HUL]**, *Object Oriented Specification Style in LOTOS*, Wilfred H. P. van Hulzen, LOTOSPHERE LO/WP1/T1.1/RNL/N00002, July 1989.
- **[KOH]**, *An introduction to neural computing*, Kohonen, in: 'Neural Networks', vol 1, number 1, 1988.
- **[KOI]**, *MIND: A specification formalism for neural networks*, P. Koikkalainen, in: 'Artificial Neural Networks' (ed. T. Kohonen et al.), North Holland, 1991.
- **[MIL]**, *A Calculus of Communicating Systems*, R. Milner, Springer-Verlag, 1980.
- **[RUD]**, *Inheritance in LOTOS*, S. Rudkin, in: 'Formal Description Techniques IV' (ed. K. Parker and G. Rose), North Holland 1991.
- **[RUM]**, *Parallel Distributed Processing — explorations in the microstructure of cognition. Vol 1: foundations*, D.E Rumelhart et al., MIT Press, 1987.
- **[SMI]**, *A framework for nural network specification*, L. S. Smith, IEEE Transactions on software engineering, vol. 18, no. 7, July 1992.
- **[TUR]**, *SPLICE I: Specification using LOTOS for an interactive customer environment — phase 1*, K.J. Turner, University of Stirling Technical Document, 1992.
- **[ZIP]**, *P3: A parallel network simulation system*, D. Zipser, in: 'Neural Networks'.

A The Perceptron (Generation-Simulation) Model

specification NETWORK [inn, out, linkin, linkout, addlink, addnode, complete, update]:**noexit**
library Boolean, NaturalNumber **endlib**
type Ident **is** Boolean **sorts** Ident_Sort
opns

```
        Base :-> Ident_Sort
        _eq_, _lt_ : Ident_Sort, Ident_Sort -> Bool
        Next : Ident_Sort -> Ident_Sort
```

```
eqns ... endtype (* Ident *)
type Node is Ident renamedby sortnames Node for Ident_Sort
opnnames N0 for Base N_N for Next
endtype (* Node *)
type SorW is sorts SorW
opns

        Strengthen, Weaken: -> SorW

endtype (* SorW *)
type Num is Boolean sorts Num
opns

        zero, one: -> Num succ: Num -> Num
        add,sub: Num, Num -> Num _ge_, _lt_: Num, Num -> Bool

eqns ... endtype (* Num *)
type NodeIDSet is Boolean,Node sorts NodeIDSet
opns

        NoNodes :-> NodeIDSet addnode : NodeIDSet, Node -> NodeIDSet
        is_node: NodeIDSet, Node -> Bool is_empty: NodeIDSet ->Bool
        remove: NodeIDSet,Node -> NodeIDSet

eqns ... endtype (* NodeIDSet *)
type NodeState is Node, NodeIDSet, Num sorts NodeState
opns

        NS: Node, NodeIDSet, NodeIDSet, Num, Num -> NodeState
        addto, addfrom: NodeState, Node -> NodeState
        getID: NodeState -> Node getFr, getTo: NodeState -> NodeIDSet
        getTh, getWe: NodeState -> Num

eqns ... endtype (* NodeState *)
type NetState IS NodeState, Boolean sorts NetState
opns

        None:-> NetState addnodestate: NetState, NodeState -> NetState
        addlink: NetState, Node, Node -> NetState getnode: NetState -> NodeState
        remove: NetState -> NetState IsEmpty: NetState -> Bool

eqns
endtype (* NetState *) behaviour
NetSpec[inn,out,linkin,linkout,addlink,addnode,complete,update] (NoNodes,NoNodes)
where(*————————————————————————————————————————*)
process NetSpec[inn,out,linkin,linkout,addlink,addnode,complete,update]
(InSet:NodeIDSet,OutSet:NodeIDSet):noexit:= hide message in
( NNET [addlink,addnode,complete,message,update](N_N(N0),None)
| [linkin,linkout,complete,addlink,message] |
InOut[inn,out,linkin,linkout,addlink,complete,message](INSet, OutSet))
where(*————————————————————————————————*)
process InOut[inn,out,linkin,linkout,addlink,complete,message]
(InS:NodeIDSet, OS:NodeIDSet): noexit:=
(linkin?Node1:Node; addlink!N0!Node1;
InOut[inn,out,linkin,linkout,addlink,complete,message]
(addnode(InS,Node1),OS))[]
(linkout?Node1:Node; addlink!Node1!N0;
InOut[inn,out,linkin,linkout,addlink,complete,message]
(InS,addnode(OS,Node1)))[]
(addlink?Node1:Node?Node2:Node
[(not(is_node(InS,Node2))) and (not(is_node(OS,Node1)))];
InOut[inn,out,linkin,linkout,addlink,complete,message] (InS,OS))[]
(complete; ModelIO[inn,out,message](InS,OS))
where (*————————————————————————————————*)
process ModelIO[inn,out,message]
(InS:NodeIDSet,OS:NodeIDSet):noexit:=
(GetIns[inn,message](InS,OS,InS) >>
(GiveOuts[message,out](InS,OS,OS) >>
ModelIO[inn,out,message](InS,OS)))
where (*————————————————————————————————*)
process GetIns[inn,message]
(InS:NodeIDSet,OS:NodeIDSet,ToGet:NodeIDSet) :exit:=
([is_empty(ToGet)] -> exit)[]
([not(is_empty(ToGet))] ->
((inn?NodeIn:Node!one[is_node(ToGet,NodeIn)]; message!N0!one;
GetIns[inn,message](InS,OS,remove(ToGet,NodeIn)))[]
(inn?NodeIn:Node!zero[is_node(ToGet,NodeIn)];
message!N0!zero;
GetIns[inn,message](InS,OS,remove(ToGet,NodeIn)))))
endproc (* GetIns*)
process GiveOuts[message,out]
(InS:NodeIDSet,OS:NodeIDSet,ToGive:NodeIDSet) :exit:=
([is_empty(ToGive)] -> exit)[]
([not(is_empty(ToGive))] -> ((message?fr:Node?N0:Node[is_node(OS,fr)]; out!fr!one;
GiveOuts[message,out](InS,OS,remove(ToGive,fr)))[]
(message?fr:Node?to:Node[not(is_node(OS,fr))];
GiveOuts[message,out](InS,OS,ToGive))))
```

```
endproc (* GiveOuts*)
endproc (* ModelIO *)
endproc (* InOut *)
process NNET[addlink,addnode,complete,message,update] (ID:Node,NS1:NetState):noexit:=
((addlink?from:Node?to:Node[((from lt ID)and(to lt ID))and(not(from eq to))];
NNet[addlink,addnode,complete,message,update] (ID,addlink(NS1,from,to))) []
(addnode?Th:Num?We:Num;
NNet[addlink,addnode,complete,message,update]
(N_N(ID),addnodestate(NS1,NS(ID,NoNodes,NoNodes,Th,We)))) []
(complete;(MakeNet[message,update](NS1))))
where (*————————————————————————————*)
process MakeNet[message,update](NS1:NetState): noexit:= hide madestep in
([IsEmpty(NS1)] -> stop) []
([not(IsEmpty(NS1))] ->
( NNode[message,update]
(getID(getNode(NS1)), getFr(getNode(NS1)),
getTo(getNode(NS1)), getTh(getNode(NS1)), getWe(getNode(NS1)))
| [message, update] |
(madestep; MakeNet[message,update](remove(NS1)))))
where (*————————————————————————————*)
process NNode[message,update]
(ID:Node,CommFrom:NodeIDSet,CommTo:NodeIDSet, Threshold:Num,WeightOut:Num):noexit:=
hide MessageOn in
( NodeInt[message, MessageOn]
(ID,CommFrom,CommTo)
| [MessageOn] |
Node[MessageOn,update](ID,CommFrom,CommTo,Threshold,WeightOut))
where (*————————————————————————————*)
process NodeInt[message,MessageOn] (ID:Node,CF:NodeIDSet,CT:NodeIDSet): noexit:=
(message?from:Nat?ID:Node?Am:Num; ((MessageOn!from!ID!Am; exit) |||
NodeInt[message,MessageOn](ID,CF,CT))) []
(message?from:Node?to:Node?Am:Num[not(to eq ID)];
NodeInt[message,MessageOn](ID,CF,CT)) []
(MessageOn!ID?to:Node?Am:Num[not(to eq ID)]; ((message!ID!to!Am; exit) |||
NodeInt[message,MessageOn](ID,CF,CT)))
endproc (* NodeInt *)
process Node[mess, update]
(ID:Node,Froms:NodeIDSet,Tos:NodeIDSet,Threshold:Num,WeightOut:Num): noexit:=
GetInMessages[mess,update]
(ID,Froms,Tos,Threshold,WeightOut,Froms,zero)
where (*————————————————————————————*)
process GetInMessages[mess,update]
(ID:Node,Froms:NodeIDSet,Tos:NodeIDSet,Threshold:Num,WeightOut:Num,ToGet: NodeI-
DSet,Total:Num): noexit:= hide gotmessages in
(([is_empty(ToGet)] -> (gotmessages;
SendOutMessages[mess,update](ID,Froms,Tos,Threshold,WeightOut,Total))) []
([not(is_empty(ToGet))] -> (mess?from:Node!ID?Am:Num[is_node(ToGet,from)];
GetInMessages[mess,update] (ID,Froms,Tos,Threshold,WeightOut,remove(ToGet,from),add(Total,Am)))))
where (*————————————————————————————*)
process SendOutMessages[mess,update]
(ID:Node,Froms:NodeIDSet,Tos:NodeIDSet,Threshold:Num,WeightOut:Num,Total:Num):
noexit:=
hide nofires,fires in
( ([Threshold ge Total] -> (nofires; (Send[mess,update](ID,Tos,zero) >>
(update?SW:SorW; Node[mess,update](ID,Froms,Tos,Threshold,WeightOut)))))[]
([Threshold lt Total] -> (fires; (Send[mess,update](ID,Tos,WeightOut) >>
((update!strengthen; Node[mess,update](ID,Froms,Tos,Threshold,add(WeightOut,Total)))
[]
(update!weaken; Node[mess,update](ID,Froms,Tos,Threshold,sub(WeightOut,Total)))))))
where(*————————————————————————————-*)
process Send[mess,update](ID:Node,Tos:NodeIDSet,Out:Num):exit:=
([is_empty(Tos)] -> exit) [] ([not(is_empty(Tos))] ->
(mess!ID?to:Node!Out[is_node(Tos,to)];
Send[mess,update](ID,remove(Tos,to),Out)))
endproc (* Send *) endproc (* SendOutMessages *) endproc (* GetInMessages *)
endproc (* Node *) endproc (* NNode *) endproc (* MakeNet *)
endproc (* NNET *) endproc (* NetSpec *) endspec (* NETWORK *)
```