NSL/ASL: Distributed Simulation of Modular Neural Networks

Alfredo Weitzenfeld, Oscar Peguero, Sebastián Gutiérrez

Departmento Académico de Computación
Instituto Tecnológico Autónomo de México (ITAM)
Río Hondo #1, San Angel Tizapán, CP 01000
México DF, MEXICO
email: alfredo@lamport.rhon.itam.mx

Abstract. As neural systems become large and complex, sophisticated tools are needed to support effective model development and efficient simulation processing. Initially, during model development, rich graphical interfaces linked to powerful programming languages and component libraries are the primary requirement. Later, during model simulation, processing efficiency is the primary concern. Workstations and personal computers are quite effective during model development, while parallel and distributed computation become necessary during simulation processing. We give first an overview of modeling and simulation in NSL together with a depth perception model example. We then discuss current and future work with the NSL/ASL system in the development and simulation of modular neural systems executed in a single computer or distributed computer network.

Keywords: Neural Networks, Simulation, Modular, Distributed

1 Introduction

The first task in modeling a neural system is creating a desired neural architecture. Creating a neural architecture involves choosing appropriate data representations for neural components while specifying their corresponding interconnections and full network dynamics. Additionally, network input and control parameters are selected. When modeling biological systems designed to reproduce anatomical and physiological data in a faithful way, there are many ways to characterize a neuron. The complexity of the neuron depends primarily on the accuracy needed. For example, compartment models [1] are quite successful in modeling detailed electric transmission in neurons. When behavioral analysis is desired, the neural network as a whole may often be adequately analyzed using simpler neuron models such as the analog leaky integrator model [2]. And sometimes even simpler neural models are enough, as the discrete McCulloch-Pitts binary model [3] where a neuron is either on or off during each time step. The last two neural models are quite appropriate when modeling analog or binary artificial networks, respectively.

A large number of simulation systems have been developed to support different types of neural modeling [4][5]. Among these, the most outstanding at the single neuron level are GENESIS [6] and NEURON [7] specifically designed to model cells taking into account their detailed morphology. For large-scale neural networks, simulators such as Aspirin/MIGRAINES [8] and NSL - Neural Simulation Language [9] have been developed to support both biological and artificial systems.

2 Neural Modeling

While neural modeling is carried out at many levels of granularity, we emphasize in NSL modularity at each of these levels. In such a way, a neural architecture is described at the highest level of abstraction in terms of multiple *neural modules* representing the overall neural functionality in the model specified each in terms of their underlying *neural networks* implementations. Thus, a complete neural architecture would consist of (1) a set of interconnected neural modules and (2) a set of interconnected neurons specifying each neural module.

2.1 Neural Modules

Neural modules in NSL are hierarchically organized in a tree structure where a root module may be further refined into additional *submodules* as shown in Figure 1. The hierarchical module decomposition results in *module assemblages* – a network of submodules seen in their entirety in terms of a single higher-level neural module. Modules (and submodules) may be implemented independently from each other in both a top-down and bottom-up fashion, an important benefit of modular design.

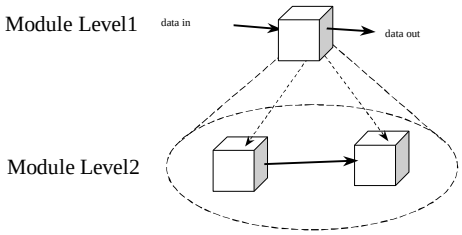

Fig. 1. NSL computational model is based on hierarchical interconnected modules. A module at a higher level (level 1) is decomposed (dashed lines) into two interconnected (solid arrow) submodules (level 2).

The actual module representation includes a set of unidirectional input and output *data ports* supporting external communication between modules. These *ports* represent module entry or exit points, used to receive or send data from or to other modules, as shown in Figure 2.

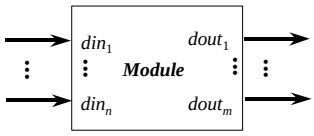

Fig. 2. Each Module may contain multiple input, $din_1,...,din_n$, and output, $dout_1,...,dout_m$, data ports for unidirectional communication.

2.2 Neural Networks

Each neural module is implemented by its underlying neural network and may recruit any number of neurons for its implementation, as shown in Figure 3.

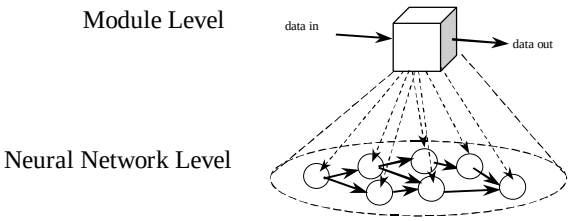

Fig. 3. A module encapsulates a neural network. Although neurons could be treated themselves as modules for further refinement, we treat them as separate entities, thus drawing them as spheres instead of cubes.

While different neural models have been simulated with NSL, we consider at the highest level a "simple" neuron having its internal state described by a single scalar quantity or membrane potential *mp*, input *s* and output *mf*, specified by some nonlinear function, as shown in Figure 4.

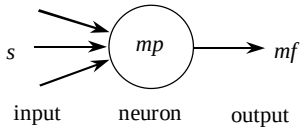

Fig. 4. Single compartment neural model represented by its membrane potential *mp*, and firing *mf*. *s* represents the set of inputs to the neuron.

The neuron may receive input from many different neurons, with only a single output branching to many other neurons. The choice of transformation from *s* to *mp* defines the particular neural model utilized. The *leaky integrator* model is described by

$$t\,\frac{dmp(t)}{dt} = -mp(t) + s(t)$$

where the *average firing rate* or output of the neuron, *mf*, is obtained by applying some "threshold function" to the neuron's membrane potential where *s* is usually described by a non-linear function also known as a *threshold* function, such as *ramp*, *step*, *saturation* or *sigmoid*,

$$mf(t) = s\,(mp(t))$$

The neural network itself is made of any number of interconnected neurons, where the most common formula for the input to a neuron is

$$sv_j = \sum_{i=0}^{n-1} w_i uf_i(t)$$

where $uf_i(t)$ represents the firing of neuron u_i whose output is connected to the jth input line of the neuron v_j, and w_i is the weight on that link (*up* and *vp* are analogous to *mp*, while *uf* and *vf* are analogous to *mf*).

2.3 Depth Perception Model

In depth perception, a three dimensional scene presented to the left eye differs from that presented to the right eye. A single point projection on each retina corresponds to

a whole ray of points at different depths in space, but points on two retinae determine a single depth point in space, the intersection of the corresponding projectors, as shown in Figure 5.

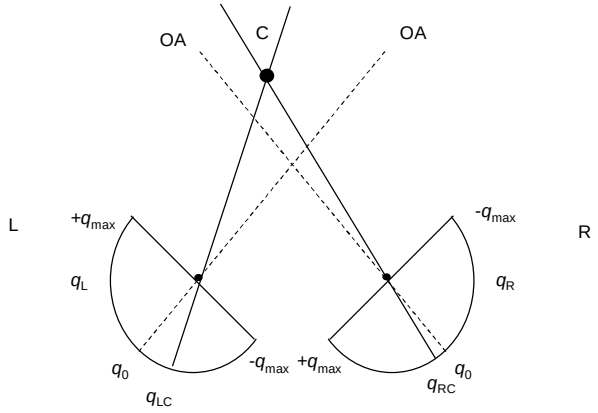

Fig. 5. Target C in space is projected to the two retinae L and R at different *disparity* points, q_{LC} and q_{RC}.

A particular problem in depth perception is that of ghost targets, where for example as shown in Figure 6, the projections for targets A and B correspond to the same projections from ghost targets C and D resulting in ambiguities.

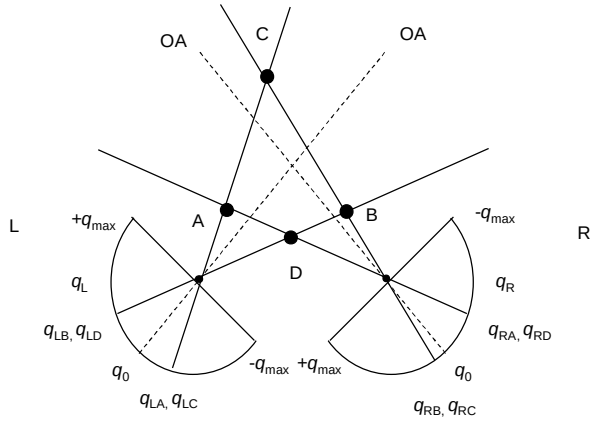

Fig. 6. Targets A and B generate the same projections as ghost targets C and D in the two retinae.

The depth perception model developed by House [10] uses two systems to build a depth map, one driven by *disparity* cues while the other is driven by *accommodation* cues. The accommodation driven field receives information about focal length and - left to its own devices - sharpens up that information to yield depth estimates. The disparity driven-field receives difference in retina projection to suppress ghost targets.

Both accommodation and disparity depth maps are described by a similar neural network as shown in Figure 7.

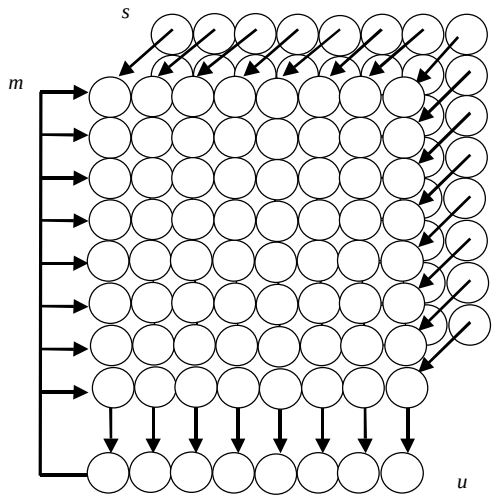

Fig. 7. Depth map neural network made of three layers *s*, *m* and *u*, where each layer is made of a set of homogeneous neurons spatially organized as shown.

The equations for the network are as follows:

$$t_m \frac{\partial m_{ij}}{\partial t} = -m_{ij} + w_m * f(m_{ij}) - w_u * g(u_j) - h_m + s_{ij}$$

$$t_u \frac{\partial u_j}{\partial t} = -u_j + \sum_i f(m_{ij}) - h_u$$

$$f(m_{ij}) = step(m_{ij}, k)$$

$$g(u_j) = ramp(u_j)$$

At the highest level each neural network corresponds to a single neural module as shown in Figure 8.

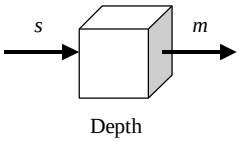

Fig. 8. Each depth map neural network is stored as a single neural module with input *s* and output *m*.

The two depth systems are intercoupled so that a point in the accommodation field excites the corresponding point in the disparity field, and viceversa. This intercoupling is shown at the neural network level in Figure 9.

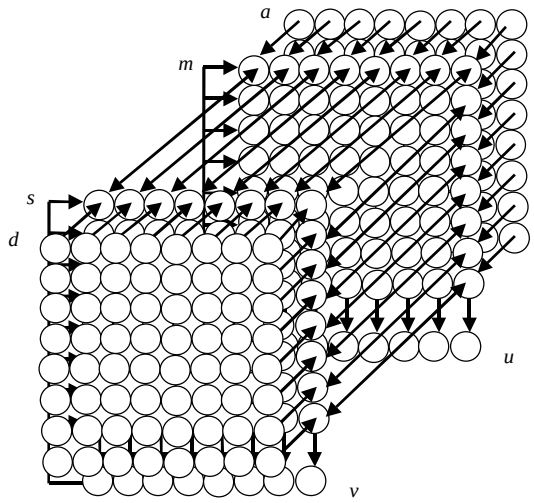

Fig. 9. Intercoupled accommodation and disparity depth maps made of layers *a*, *m* and *u*, and layers *d*, *s* and *v*, respectively.

The equations for the disparity depth map are as follows:

$$t_s \frac{\partial s_{ij}}{\partial t} = -s_{ij} + w_s * f(s_{ij}) + w_t * f(t_{ij}) - w_v * g(v_j) - h_s + d_{ij}$$

$$t_v \frac{\partial v_j}{\partial t} = -v_j + \sum_i f(s_{ij}) - h_v$$

$$f(s_{ij}) = sigma(s_{ij})$$

$$g(v_j) = ramp(v_j)$$

The equations for the accommodation depth map are as follows:

$$t_m \frac{\partial m_{ij}}{\partial t} = -m_{ij} + w_m * f(m_{ij}) + w_t * f(t_{ij}) - w_u * g(u_j) - h_m + a_{ij}$$

$$t_u \frac{\partial u_j}{\partial t} = -u_j + \sum_i f(m_{ij}) - h_u$$

$$f(m_{ij}) = sigma(m_{ij})$$

$$g(u_j) = ramp(u_j)$$

The corresponding NSL model consists of three interconnected modules: *Retina* *r*, *Depth* (accommodation) *m* and (disparity) *s*, as shown in Figure 10. (Note that *Retina* and *Depth* correspond to module types, while *r*, *m* and *s* represent module instantiations, where *m* and *s* have similar type definition).

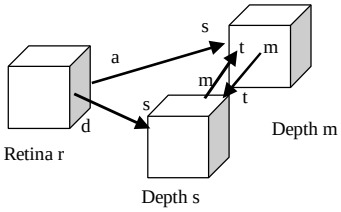

Fig.10. The *Retina* module contains two output ports, *dp* and *ap*, for disparity and accommodation, respectively. The *Depth* module consists of an input port *sp*, receiving data from the *Retina*, a second input port *tp*, receiving input from the other *Depth* module, and an output port *mp*.

3 Neural Simulation

Simulation of neural networks varies depending on whether it relates to artificial or biological systems. Artificial neural networks particularly those involving learning usually require a two-stage simulation process, (1) an initial training phase and (2) a subsequent processing or running phase. On the other hand, biological networks usually require a single running phase. The simulation process involves interactively specifying aspects of the model that tend to change, such as network input and parameter values, as well as simulation control and visualization. In general, the user analyzes output generated by the simulation, both visual and numerical, deciding if any modifications are necessary. If network input or parameter changes are necessary, these may be interactively specified, and the model is simulated again with the newly specified changes. On the other hand, if structural modifications in the neural architecture are required, then the developer must go back to the modeling phase.

In terms of execution, concurrency plays an important role in neural network simulation, not only as a way to increase processing performance but also to model neurons more faithfully [11]. Yet, most simulation systems execute sequentially due to lower complexity in their underlying hardware and software. On the other hand, distributed processing can be a more effective solution to parallel processing requiring the use of more sophisticated and expensive multiprocessor computers as opposed to a network of "standard" computers in the former case. The NSL computational model enables both sequential and distributed processing (as well as parallel processing) with very limited overhead from the user's side.

3.1 Sequential Simulation

Under sequential neural network simulation, neural dynamics are executed one equation at the time where the output from one neuron is immediately fed as input to the next one. In NSL, the same procedure is applied to modules where each module would get executed in a predetermined order sending newly generated output port values immediately to their corresponding interconnected input ports. This approach simplifies simulation results in that the order of computation depends directly from the order of computation. To further simulate brain-like concurrent behavior while doing sequential processing, NSL interleaves module execution while buffering output

from one module to the next one, making processing order unimportant. Yet, this approach does not improve processing efficiency.

3.2 Distributed Simulation

The *ASL - Abstract Schema Language* [12] is a distributed architecture integrating with the NSL simulation system to support both single processor and multiprocessor distributed computation. (The term *schema* [13] in ASL corresponds to *active* or concurrent objects [14] with the ability to process concurrently.) ASL extends the NSL module and port interconnection model adding *asynchronous* inter-process communication, especially suitable to optimize communication performance. In such an environment, all simulation control as well as graphical displays take place in the *console* in one of the machines responsible for distributing processing among local and/or remote machines. The actual distribution is specified by the user or automatically generated by the system depending on the available machines in the network. In ASL, module execution is managed by a *Thread Manager* in each process, while the *IPC* (*Inter-Process Communication*) *Manager* is responsible for external data communication. The current implementation of ASL is in C++ using *PVM* [15] as the IPC Manager and *PThreads* [16] as the Thread Manager.

3.3 Depth Perception Simulation

Under both sequential and distributed environments, simulation of the depth perception model generates similar results although with different efficiency. In Figure 11 we show the initial input to the model. Figure 12 shows two different targets with the corresponding initial accommodation (top right) and disparity (bottom right) depth maps.

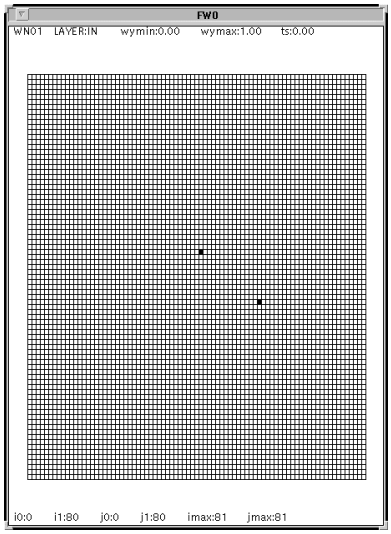

Fig. 11. The *Retina* module contains two output ports, *dp* and *ap*, for disparity and accommodation, respectively. The *Depth* module consists of an input port *sp*, receiving data from the *Retina*, a second input port *tp*, receiving input from the other *Depth* module, and an output port *mp*.

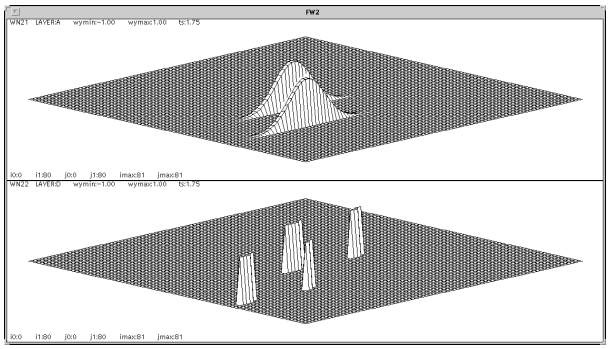

Fig. 12. The *Retina* module contains two output ports, *dp* and *ap*, for disparity and accommodation, respectively. The *Depth* module consists of an input port *sp*, receiving data from the *Retina*, a second input port *tp*, receiving input from the other *Depth* module, and an output port *mp*.

Two snapshots taken during model simulation are shown in Figure 13, for t=0.50 at the beginning of the simulation and Figure 14, for t=1.75 at the end. The top portions of the figures represent accommodation modules while the bottom portions represent disparity modules. As time progresses, target ghost start to disappear and only real activity common to both accommodation and disparity stay active.

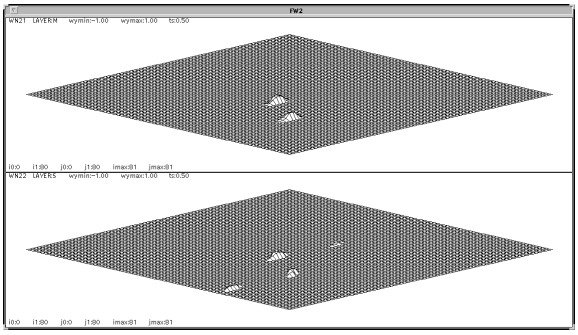

Fig. 13. Disparity *s* and accommodation *m* modules during simulation time 0.50.

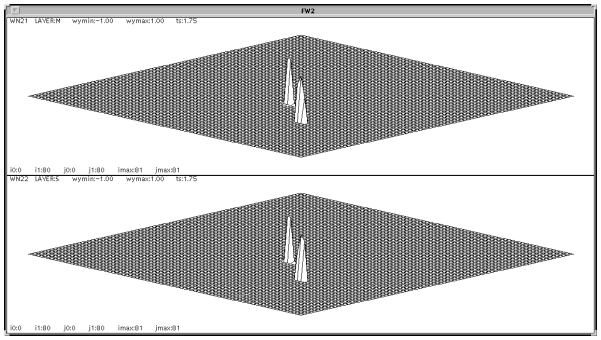

Fig. 14. Disparity *s* and accommodation *m* modules during simulation time 1.75.

For this particular simulation, the distribution architecture for the depth perception model consisted of three Unix workstations, one used as console displaying graphics and providing user interactivity while the other two processed the actual modules, as shown in Figure 15.

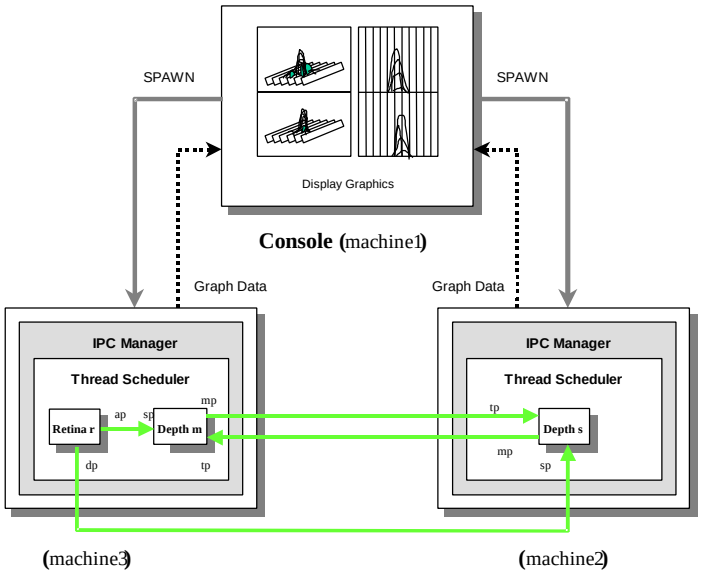

Fig. 15. Distributed simulation of the depth perception model under NSL/ASL.

Current benchmarks have shown that the distributed simulation of the Depth Perception model may run between 5 to 10 times as fast as the sequential one depending on overall machine and network load.

5 Conclusions

This paper has shown the basic concepts involved in modular neural network development and distributed simulation in NSL/ASL. While a large number of neural models have been developed throughout the years in NSL, we are in the process of testing them under the NSL/ASL distributed architecture. Distributed simulation performance, to be made more precise as further benchmarks are carried out, should reflect good processing improvement not only in smaller models but especially in larger models as well. Additionally, we are in the process of putting together independently developed modules as part of more complex models involving a large number of modules. Previously, this was prohibitively expensive to either model or simulate, particularly with workstations. For example, the depth perception model becomes just one component in the Frog's *Learning to Detour* model [17].

Other simulation systems have taken "parallel" approaches to deal with complex simulation, such as PGENESIS, NEURON[1], as well as previous versions of NSL. These versions tended not to be too "friendly" in their usage requiring special effort in

[1] These two implementations are accessible from the Pittsburgh Supercomputer Center.

the part of the modeler to adapt to the programming environment. On the other hand, the distributed NSL/ASL architecture provides a more accessible alternative to standalone parallel computation. Furthermore, in more sophisticated distributed environments a supercomputer can play a key role intensive module processing with graphics and command interaction taking place in a workstation.

Future extensions to the NSL/ASL distributed system involve how to troubleshoot it as well as optimize it. A reflective meta-level architecture [18] is being added to ASL providing monitoring capabilities [19] at the module and port communication level to improve overall simulation performance, *e.g. scheduling, deadlocks, load balancing, communication*, etc. Other extensions to NSL/ASL include linkage to the NSL Java version in a single heterogeneous Java/C++ distributed environment where we are considering other alternatives to PVM (as well as Pthreads) for inter-process communication particularly CORBA [20]. And while distributed computation is quite useful in improving processing performance, applications such as the control of remote robots [21] or the Web itself, have distribution as a primary requirement [22][2].

Ackowledgments

We thank the NSF-CONACyT collaboration grant (#IRI-9522999 in the US and #546500-5-C018-A in Mexico), the CONACyT REDII grant in Mexico, as well as the "Asociación Mexicana de Cultura, A.C.". Additionally we want to thank all students previously involved in the project in particular Salvador Mármol and Claudia Calderas.

References

1. Rall, W., Branching dendritic trees and motoneuron membrane resistivity, Exp. Neurol., 2:503-532, 1959.
2. Arbib, M.A., *The Metaphorical Brain 2: Neural Networks and Beyond*, pp. 124-126. Wiley Interscience, 1989.
3. McCulloch, W.S. & Pitts, W.H., A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bull. Math. Biophys*. 5:115-133, 1943.
4. De Schutter, E., A Consumer Guide to Neuronal Modeling Software. *Trends in Neuroscience*. 15:462-464, 1992.
5. Murre, J., Neurosimulators, in Michael Arbib (ed.) *The Handbook of Brain Theory and Neural Networks*, pp. 634-639, MIT Press, 1995.
6. Bower, J.M. & Beeman, D., *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*, TELOS/Springer-Verlag, 1998.
7. Hines, M.L. & Carnevale N.T., The NEURON simulation environment. *Neural Computation*, 9:1179-1209, 1997.

[2] All information regarding NSL and ASL as well as C++ downloads are available from "http://cannes.rhon.itam.mx" while the NSL Java version is obtainable from "http://www-hbp.usc.edu".

8. Leigthon, R.R. & Wieland, A.P. , The Aspirin/Migraines Software Package, in J. Skrzypek (ed.) *Neural Network Simulation Environments*, pp. 209-227, Kluwer, 1994.
9. Weitzenfeld, A., Alexander, A. & Arbib, M.A., *The Neural Simulation Language NSL, Theory and Applications*, MIT Press, 2000.
10. House, D., Depth Perception in Frogs and Toads: A study in Neural Computing, *Lecture Notes in Biomathematics 80*, Springer-Verlag, 1985.
11. Weitzenfeld, A. & Arbib, M., A Concurrent Object-Oriented Framework for the Simulation of Neural Networks, in Proc. of ECOOP/OOPSLA '90 Workshop on Object-Based Concurrent Programming, *OOPS Messenger*, 2(2):120-124, April, 1991.
12. Weitzenfeld, A., ASL: Hierarchy, Composition, Heterogeneity, and Multi-Granularity in Concurrent Object-Oriented Programming, *Proc. Workshop on Neural Architectures and Distributed AI: From Schema Assemblages to Neural Networks*, Oct 19-20, USC, October 19-20, 1993.
13. Arbib, M.A., Schema Theory. In Stuart Shapiro (ed.), *The Encyclopedia of Artificial Intelligence*, 2nd. Edition, 2:1427-1443, Wiley, 1992.
14. Yonezawa, A. & Tokoro, M., *Object-oriented concurrent programming*, MIT Press, 1987.
15. Geist, A., Beguelin, A., Dongarra J., Jiang, W., Mancheck, R. & Sunderam, V., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
16. Lewis B. & Berg D.J., *Multithreaded Programming with Pthreads*, Sun Microsystems Press - Prentice Hall, 1998.
17. Corbacho, F. & Arbib, M.A., Learning Internal Models to Detour, *Society for Neuroscience*. Abs. 624.7, 1997.
18. Kiczales, G. & Paepcke, A., *Open Implementations and Metaobject Protocols*, Palo Alto Research Center, Xerox Corporation, 1996.
19. Gutiérrez, S. *Design of a Reflective Meta-level Architecture for NSL/ASL*, TR-99-01, Dept. Comp Eng, ITAM, Sept., 1999.
20. Mowbray, T. & Ruh, W., *Inside CORBA: Distributed Object Standards and Applications*, Addison-Wesley, 1998.
21. Weitzenfeld, A., Arkin, R.C., Cervantes-Perez, F., Olivares, R., & Corbacho, F., A Neural Schema Architecture for Autonomous Robots, *Proc. 1998 Int. Symposium on Robotics and Automation*, Dec. 12-14, Saltillo, Coahuila, Mexico, 1998.
22. Alexander, A., Arbib, M.A. & Weitzenfeld, A., Web Simulation of Brain Model, in A. Bruzzone, A. Uhrmacher and E. Page (eds.) Proc. 1999 Int. Conf. On Web-Based Modeling and Simulation , pp. 124-126. 31(3):29-33, Soc. Comp Sim, 1999.