CHAPTER 2.2

NSL Neural Simulation Language

Michael A. Arbib, Amanda Alexander, and Alfredo Weitzenfeld
University of Southern California Brain Project and Computer Science Department,
University of Southern California, Los Angeles, California

The NSL Neural Simulation Language provides a platform for building neural architectures (modeling) and for executing them (simulation). NSL is based on object-oriented technology and provides modularity at all model development levels. In this chapter, we discuss these basic concepts and how NSL takes advantage of them. NSL, now in its third major release, is a neural network simulator which is both general and powerful, designed for users with diverse interests and programming abilities.

For novice users interested only in an introduction to neural networks, we provide user-friendly interfaces and a set of predefined artificial and biological neural models. For more advanced users well acquainted with the area who require more sophistication, we provide evolved visualization tools together with extensibility and scalability. We provide support for varying levels in neuron model detail, which is particularly important for biological neural modeling. In artificial neural modeling, the neuron model is very simple, with network models varying primarily in their network architectures and learning paradigms. While NSL is not particularly intended to support detailed single neuron modeling, as opposed to systems such as GENESIS and NEURON primarily designed for this task, NSL does provide sufficient expressiveness to support this level of modeling.

The Neural Simulation Language (NSL) has evolved for over a decade. The original system was written in C (NSL 1) in 1989, with a second version written in C++ (NSL 2) in 1991 and based on object-oriented technology. Both versions were developed at the University of Southern California by Alfredo Weitzenfeld, with Michael Arbib involved in the overall design. The present version, NSL 3, is a major release completely restructured over former versions both as a system as well as the supported modeling and simulation, including modularity and concurrency. It provides a powerful neural development environment supporting the efficient creation and execution of scalable neural networks, incorporating a compiled language, NSLM, for model development, and a scripting language, NSLS, for model interaction and simulation control. It offers rich graphics and a full mouse- driven window interface supporting creation of new models as well as their control and visualization.

NSL 3 includes two different environments, one in Java (NSLJ, developed at USC by Amanda Alexander's team) and the other in C++ (NSLC, developed at ITAM in Mexico by Weitzenfeld's team), again with Arbib involved in the overall design. Both environments support similar modeling and simulation and are described fully in Weitzenfeld *et al*. (2001). In this chapter, we focus on NSLJ, the version developed by the USC Brain Project team and available from our Website. We offer a free download of the complete NSLJ system, including full source code as well as forthcoming new versions. We also provide free and extensive support for downloading new models from our Websites, where users may contribute with their own models and may criticize existing ones (see Chapter 6.2; http://www-hbp.usc.edu/Projects/BMW.htm). The advantages of Java include:

1. *Portability*: Code written in Java runs without changes "everywhere."
2. *Maintainability*: Java code requires maintaining one single software version for different operating systems, compilers, and platforms.
3. *Web-oriented*: Java code runs on the client side of the Web, simplifying exchange of models without the owner of the model having to provide a large server on which other people can run simulations.

In summary, NSL is especially suitable for use in an academic environment where NSL simulation and model development can complement theoretical courses in both biological and artificial neural networks and for use in a research environment where scientists require rapid model prototyping and efficient execution. NSL may easily be linked to other software tools, such as additional numerical libraries, or hardware, such as robotics, by doing direct programming in either Java or C++. NSL supports the design of modular neural networks which simplify modeling and simulation while providing better model extensibility.

2.2.1 Modeling and Simulation of Neural Networks

Neural network simulation is an important research and development area extending from biological studies to artificial applications. In this book, we stress biological neural networks designed to model the brain in a faithful way at a level of detail appropriate to the data under analysis.

Modeling and Simulation

Modeling develops a neural architecture which can explain and reproduce anatomical and physiological experimental data. It involves choosing appropriate data representations for neural components, neurons, and their interconnections, as well as network input, control parameters, and network dynamics specified in terms of a set of mathematical equations. The neuron model varies depending on the details being described. Neuron models can be very sophisticated biophysical models, such as compartmental models based on the Hodgkin-Huxley model. When behavioral analysis is desired, simpler neuron models such as the analog *leaky integrator* model may be appropriate. The particular neuron model chosen defines the dynamics for each neuron, yet a complete network architecture also involves specifying interconnections among neurons as well as specifying input to the network and choosing appropriate parameters for different tasks using the neural model specified. Moreover, many models involve learning,whose dynamics must be specified in the model architecture.

Simulation, then, consists of using the computer to see how the model behaves for a variety of input patterns and parameter settings. A simulation may use values pre-specified in the original formulation of the model but will in general involve specifying one or more aspects of the neural architecture that may be modified by the user. Simulation also involves analyzing the results, both visual and numerical, generated by the simulator; on the basis of these results, one can decide if any modifications are necessary in the network input or parameters. If changes are required, these may be interactively specified and the model re-simulated, or the changes may require more structural modifications at the neural architecture level and the model will have to be re-compiled. Simulation also involves selecting one of the many approximation methods used to solve neural dynamics specified through differential equations.

In addition, the simulation environment may need to change when moving a model from the development phase to the test phase. When models are initially simulated, good interactivity is necessary to let the user modify inputs and parameters as necessary. As the model becomes more stable, simulation efficiency is a primary concern where model processing may take considerable time—possibly hours or even days for the largest networks to process. In our system, modularity, object-oriented programming, and concurrency play an important part in building neural networks as well as in their execution. We briefly review these key ideas.

Modularity

Modularity is today widely accepted as a requirement for structuring software systems. As software becomes larger and more complex, being able to break a system into separate modules enables the software developer to better manage the inherent complexity of the overall system. As neural networks become larger and more complex, they too may become difficult to read, modify, test, and extend. Moreover, when building biological neural networks, modularization is further motivated by taking into consideration the way we analyze the brain as a set of different brain regions. The general methodology for making a complex neural model of brain function is to combine different modules corresponding to different brain regions. To model a particular brain region, we divide it anatomically or physiologically into different neural arrays. Each brain region is then modeled as a set of neuron arrays, where each neuron is described, for example, by the leaky integrator, a single-compartment model of membrane potential and firing rate. (However, one can implement other, possibly far more detailed, neural models.) For example, Fig. 1 shows the basic components in a model describing the interaction of the *Superior Colliculus (SC)* and the saccade generator of the *Brainstem* involved in the control of eye movements. In this model, each component or module represents a single brain region.

Structured models provide two benefits. The first is that it makes them easier to understand, and the second is that modules can be reused in other models. For example, Fig. 2 shows the two previous *SC* and *Brainstem* modules embedded into a far more complex model, the *Crowley-Arbib* model of basal ganglia. Each of these modules can be further broken down into submodules, eventually reaching modules that take the form of neural arrays. Fig. 3 shows how the single *Prefrontal Cortex* (*PFC*) module can be further broken down into four submodules, each a crucial brain region involved in the control of movement.

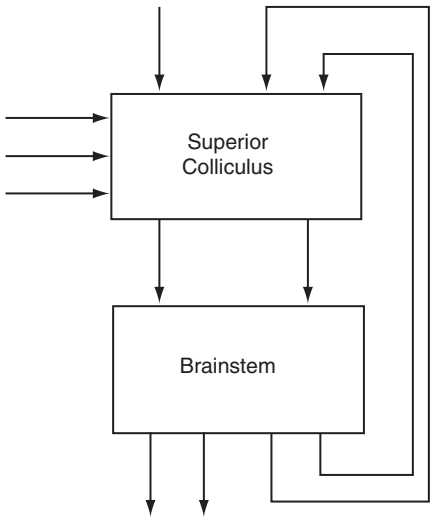

Figure 1 The diagram shows two interconnected modules, the *Superior Colliculus (SC)* and the *Brainstem*. Each module is decomposed into several submodules (not shown here), each implemented as an array of neurons identified by their different physiological response when a monkey makes rapid eye movements.

There are, basically, two ways to understand a complex system. One is to focus in on some particular subsystem, some module, and carry out studies of that in detail. The other is to step back and look at higher levels of organization in which the details of particular modules are hidden. Full understanding comes as we cycle back and forth between different levels of detail in analyzing different subsystems, sometimes simulating modules in isolation, at other times designing computer experiments that help us follow the dynamics of the interactions between the various modules. Thus, it is important for a neural network simulator to support modularization of models. This concept of modularity is best supported today by object-oriented languages and the underlying modeling concepts described next.

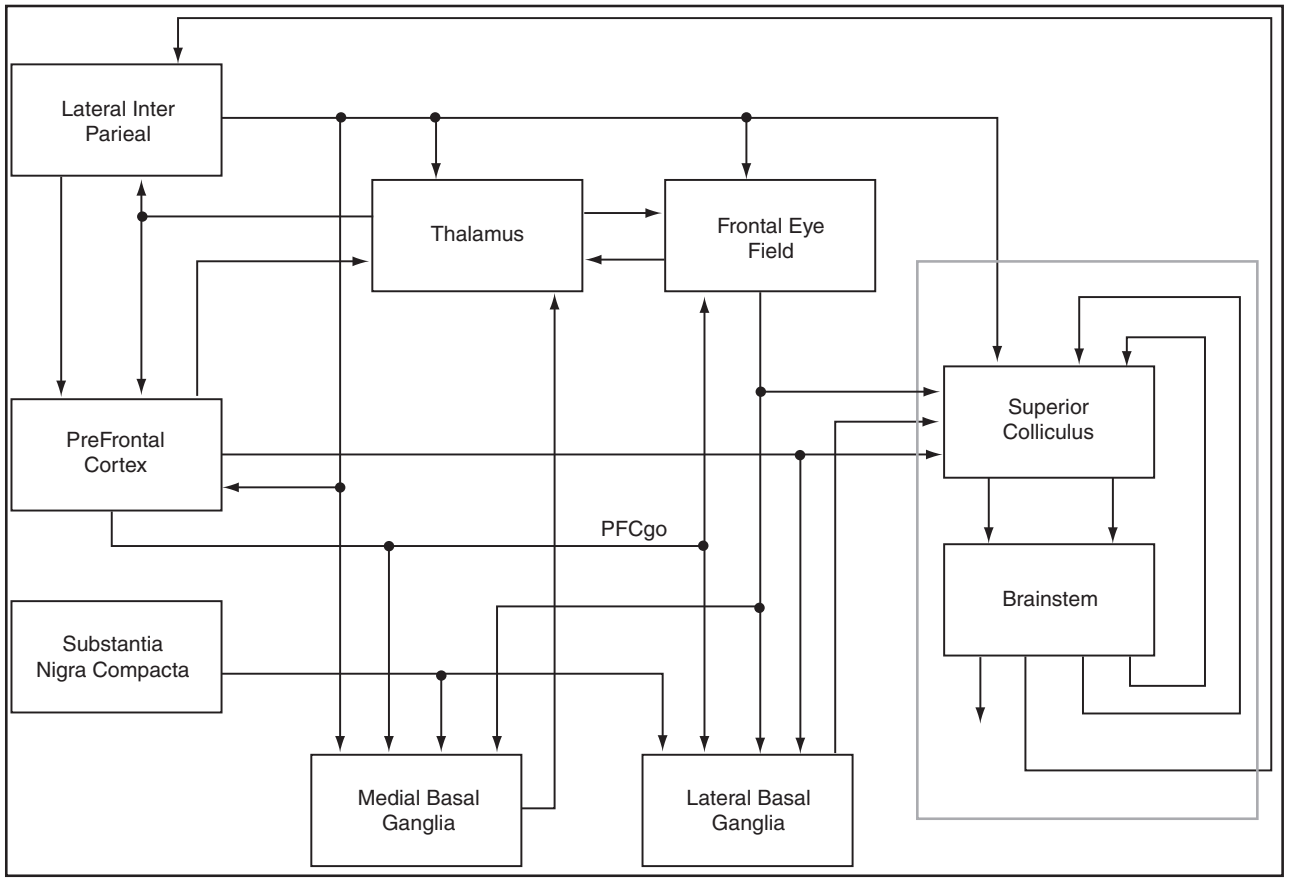

Figure 2 The diagram shows the *SC* and *Brainstem* modules from Fig. 1 embedded in a much larger model of interacting brain regions.

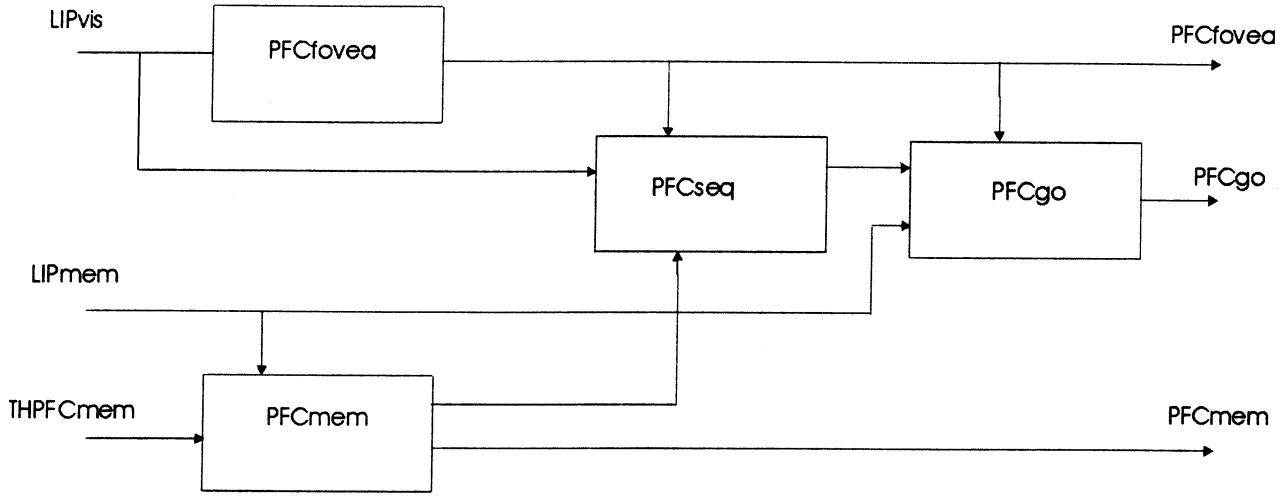

Figure 3 The *Prefrontal Cortex (PFC)* model is further decomposed into four submodules.

Object-Oriented Programming

Object-oriented technology has existed for more than 30 years; however, only in this past decade have we seen it applied in so many industries. What makes this technology special is the concept of the *object* as the basic modularization abstraction in a program. Prior to object-orientation, a complete application would be written at the data and function level of abstraction. Data and functions would be global to a program, and any changes to them could potentially affect the complete system, an undesired effect when large and complex systems are being modified. To avoid this problem, an additional level of abstraction is added—the object. At the highest level, programs are made exclusively out of objects interacting with each other through pre-defined object interfaces. At the lowest level, objects are individually defined in terms of local data and functions, avoiding global conflicts that make systems so difficult to manage and understand. Changes inside objects do not affect other objects in the system so long as the external behavior of the object remains the same. Because there is usually a smaller number of objects in a program than the total number of data or functions, software development becomes more manageable. Objects also provide abstraction and extensibility and contribute to modularity and code reuse. These seemingly simple concepts have a great repercussion in the quality of systems being built, and its introduction as part of neural modeling reflects this. Obviously, the use of object-orientation technology is only part of writing better software. How the user designs the software or neural architectures with this technology has an important effect on the system, an aspect which becomes more accessible by providing a simple to follow yet powerful modeling architecture such as that provided by NSL.

Concurrency in Neural Networks

Concurrency can play an important role in neural network simulation, both in order to model neurons more faithfully and to increase processing throughput (Weitzenfeld and Arbib, 1991). We have incorporated concurrent processing capabilities in the general design of NSL for this purpose. The computational model on which NSL is based has been inspired by the work on the Abstract Schema Language (ASL; Weitzenfeld, 1992), where *schemas* (Arbib, 1992) are *active* or concurrent objects (Yonezawa and Tokoro, 1987) resulting in the ability to concurrently process modules. NSL software is currently implemented on serial computers, emulating concurrency. Extensions to NSL and its underlying software architecture will implement genuine concurrency to permit parallel and distributed processing of modules in the near future.

2.2.2 NSL Modules and Simulation

As an object-oriented system, NSL is built with modularization in mind. As a neural network development platform, NSL provides a modeling and simulation environment for large-scale, general-purpose neural networks by the use of modules that can be hierarchically interconnected to enable the construction of very complex models. NSL provides a modeling language, NSLM, to build/code the model and a scripting language, NSLS, to specify how the simulation is to be executed and controlled. Modeling in NSL is carried out at two levels of abstraction—*modules* and *neural networks* somewhat analogous to object-orientation in its different abstraction levels when building applications. Modules define the top-level view of a model, hiding its internal complexity. A complete model in

NSL requires the following components: (1) a set of modules defining the entire model, (2) neurons comprised in each neural module, (3) neural interconnections, (4) neural dynamics, and (5) numerical methods to solve the differential equations.

Modules and Interconnections

At the highest level model architectures are described in terms of *modules* and *interconnections*. We describe in this section these concepts as well as the *model*, representing the main module in the architecture, together with a short overview of scheduling and buffering involved with modules.

Modules in NSL correspond to objects in object orientation in that they specify the underlying computational model. These entities are hierarchically organized, as shown in Fig. 4. Thus, a given module may either be decomposed into a set of smaller modules or may be a "leaf module" that may be implemented in different ways, where neural networks are of particular interest here (Fig. 5). The hierarchical module decomposition

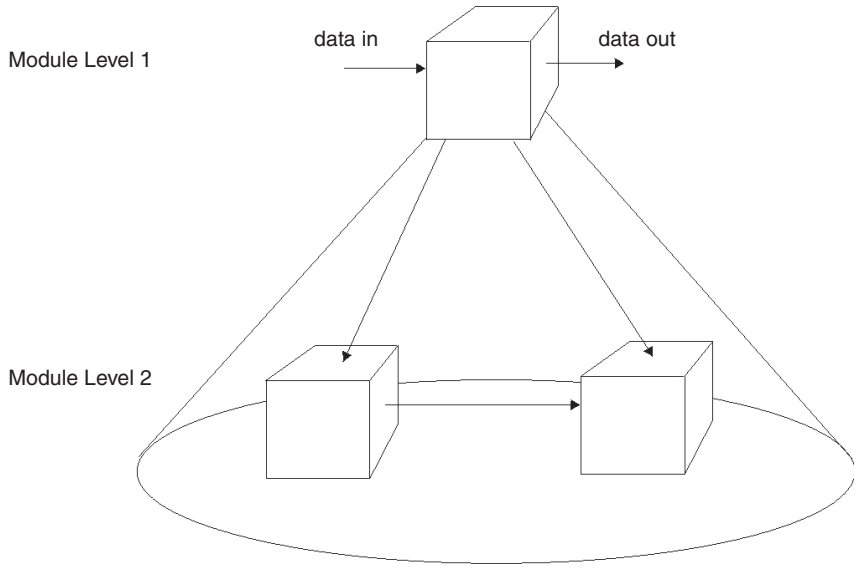

Figure 4 The NSL computational model is based on hierarchical modules. A module at a higher level (Level 1), is decomposed (dashed lines) into submodules (Level 2). These submodules are themselves modules that may be further decomposed. Solid arrows show data communication among modules. (There are also port inputs and outputs, not shown, at Level 2 which correspond to the "data in" and "data out" ports at Level 1.)

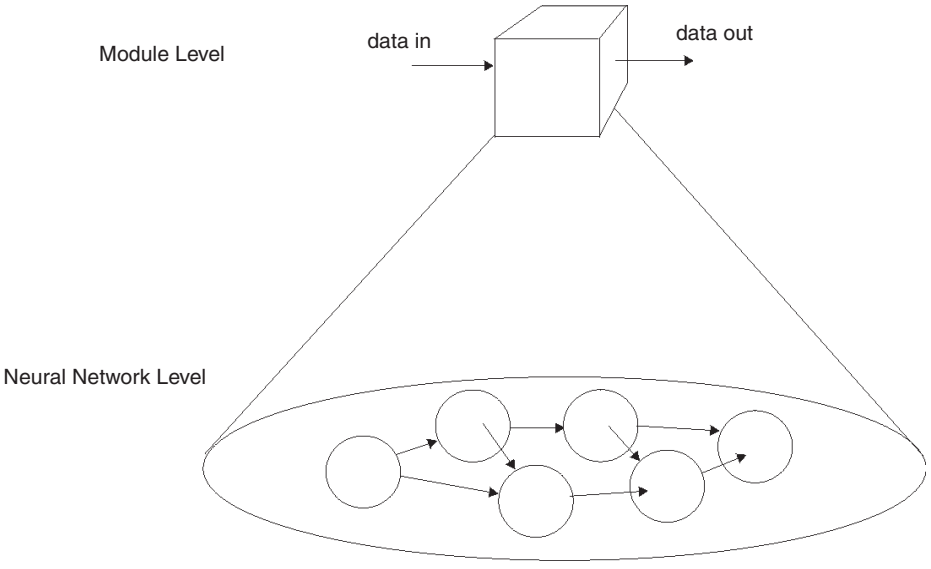

Figure 5 A module in NSL implemented by neural networks made of multiple neurons. (There are port inputs and outputs at the Neural Network Level as well, not shown, which correspond to the "data in" and "data out" ports at the Module Level.)

results in what is known as *module assemblages*—a network of *submodules* that can be seen in their entirety in terms of a single higher level module. These hierarchies enable the development of modular systems where modules may be designed and implemented independently of each other following both top-down and bottom-up development.

The *module*, the basic component in a model architecture, is somewhat analogous to the *object* in object-oriented applications. The external portion of the module is the part of the module seen by other modules. The internal portion is not seen by other modules—this makes it easier to create and modify modules independently from each other—and defines the actual module behavior. This behavior need not be reducible to a neural network; it could also be a module doing something else (e.g., providing inputs or monitoring behavior).

The most important task of a module's external interface is to permit communication between different modules. As such, a module in NSL includes a set of input and output *data ports* (we shall call them simply *ports*). The port represents an entry or exit point where data may be sent or received to or from other modules, as shown in Fig. 6. For example, the *Maximum Selector* model architecture incorporates a module having two input ports, *s_in* and *v_in*, together with a single output port, *uf*, as shown in Fig. 7.

Data sent and received through ports is usually in the form of numerical values. These values may be of different numerical types while varying in dimension. In the simplest form, a numerical type may correspond to a float, double, or integer. Dimensions correspond to a single scalar, a one-dimensional array (*vector*), a two-dimensional array (*matrix*), or higher dimensional arrays. For example, in the *Ulayer* module shown in Fig. 7, *v_in* is made to correspond to a scalar type, while *s_in* and *uf* both correspond to vector arrays (the reason for this selection will become clear very soon). In terms of implementation, the NSL *module* specification has been made as similar as possible to a *class* specification in object-oriented languages such as Java and C++ in order to make the learning curve as short as possible for those who already have a programming background. The actual code for specifying these constructs may be found in Chapter 3 of Weitzenfeld *et al*. (2001) or on our Website.

The general module definition includes the *structure* representing module attributes (data) and the *behavior* representing module methods (operations). The former specifies the ports of the module, as well as the internal variables of the module. The behavior spells out how values read in through the input ports and/or internal variables may be operated upon both to update the internal variables and to provide values as appropriate to the output ports. The *behavior* of a module will include a number of specific methods called by the simulator during model execution. These methods are used for different purposes (e.g., initialization, execution, termination).

Interconnections between modules are achieved by interconnecting output ports in one module to input ports in another module. Communication is unidirectional, flowing from an output port to an input port. Fig. 8 shows the interconnections between the two modules of a simple model, the *Maximum Selector* model, which we will describe below to introduce basic features of the NSL simulation system. In the example, a connection is made from output port *uf* in *Ulayer* to input port *v_in* in *Vlayer*; additionally, output port *vf* in *Vlayer* is connected to input port *v_in* in *Ulayer*. In general, a single output port may be connected to any number of input ports, whereas the opposite is not allowed (i.e., connecting multiple output ports to a single input port). The reason for this restriction is that the input port could receive more than one communication at any given time, resulting in inconsistencies. Again, the actual code for specifying these connections may be found in Chapter 3 of Weitzenfeld *et al*. (2001) or on our Website.

The output to input port connection is considered "same level module connectivity." By contrast, in "different level module connectivity," an output port from a module at one level is *relabeled* (we use this term instead of *connected* for semantic reasons) to an output

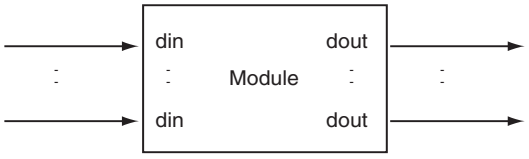

Figure 6 The NSL computational model is based on the module concept. Each module consists of multiple input ($din_1, \ldots, din_n$) and output ($dout_1, \ldots, dout_m$) *data ports* for unidirectional communication. The number of input ports does not have to be the same as the number of output ports.

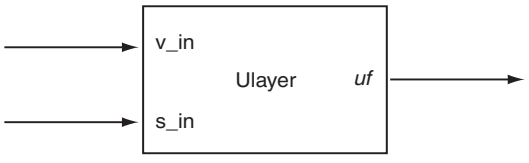

Figure 7 The ULayer module of the *Maximum Selector* model has two input ports, *s_in* and *v_in*, and a single output port, *uf*.

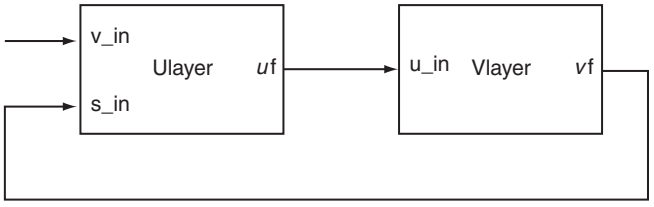

Figure 8 Interconnections between modules *Ulayer* and *Vlayer* of the *Maximum Selector* model.

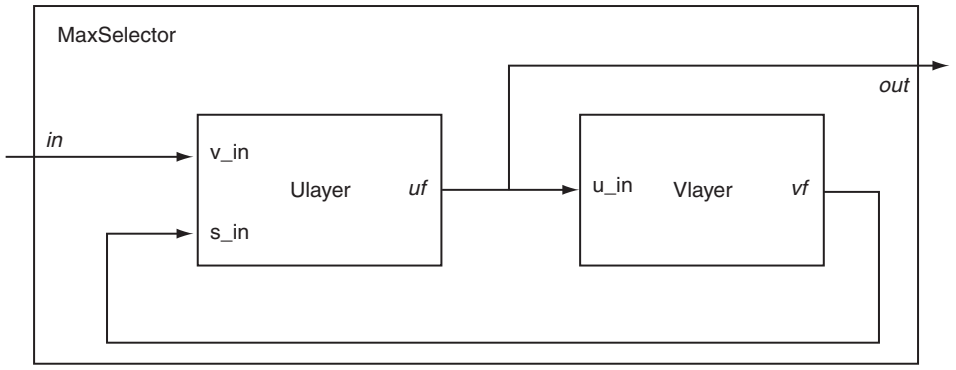

Figure 9 *Maximum Selector* model architecture contains a *MaxSelector* module with two interconnected modules, *Ulayer* and *Vlayer*.

port of a module at a different level. Alternatively, an input port at one level module may be *relabeled* to an input port at a different level. For example, in Fig. 9 we introduce the *MaxSelector* module, containing an input port *in* and an output port *out*, encapsulating modules *Ulayer* and *Vlayer*. *MaxSelector* is considered a higher level module than the other two as it contains—and instantiates—them. In general, relabeling lets input and output ports forward their data between module levels. (This supports module encapsulation in the sense that a module connected to *MaxSelector* should not connect to ports in either *Ulayer* or *Vlayer* nor be able to get direct access to any of the modules private variables.) Relabelings, similar to connections, are unidirectional, where an input port from one module may be relabeled when viewed as an input port at a different level.

We close this section with a brief mention of scheduling and buffering. *Scheduling* specifies the order in which modules and their corresponding methods are executed, while buffering specifies how often ports read and write data in and out of the module. NSL uses a multi-clock-scheduling algorithm where each module's clock may have a different time step although it is synchronized between modules during similar time steps. During each cycle, NSL executes the corresponding simulation methods implemented by the user. *Buffering* relates to the way output ports handle communication. Because models may simulate concurrency, such as with neural network processing, we have provided *immediate* (no buffer) and *buffered* port modes. In the immediate mode (sequential simulation), output ports immediately send their data to the connecting modules. In the buffered mode (pseudo-concurrent simulation), output ports do not send their data to the connecting modules until the following simulated clock cycle. In buffered mode, output ports are double buffered. One buffer contains the data that can be seen by the connecting modules during the current clock cycle, while the other buffer contains the data being currently generated that will only be seen by the connected modules during the next clock cycle.

Modules and Neural Networks

Some modules will be implemented as neural networks where every neuron becomes an element or attribute of a module, as shown in Fig. 10. Note that although neurons also may be treated as modules, they are often treated as elements inside a single module (e.g., one representing an array of neurons) in NSL. We thus draw neurons as spheres instead of cubes to highlight the latter possibility.

There are many ways to characterize a neuron. The complexity of the neuron depends on the accuracy needed by the larger model network and on the computational power of the computer being used. The GENESIS (Bower and Beeman, 1998) and NEURON (Hines and Carnevale, 1997) systems were designed specifically to support the modeling of a single neuron which takes account of the detailed morphology of the neuron in relation to different types of input. The NSL system was designed to let the user represent neurons at any level of desired detail; however, we will focus here on the simulation of large-scale properties of neural networks modeled with relatively simple neurons.

We consider the neuron shown in Fig. 10 to be "simple" as its internal state is described by a single scalar quantity, membrane potential *mp*; its input is *S*, and its output is *mf*, specified by some nonlinear function of *mf*.

The neuron may receive input from many different neurons, while it has only a single output (which may "branch" to affect many other neurons or drive the net

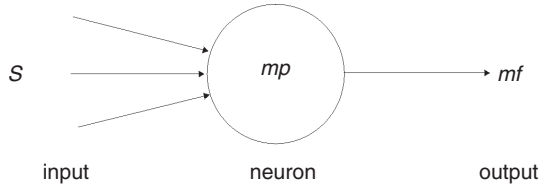

Figure 10 Single compartment neural model represented by a value, *mp*, corresponding to its membrane potential and a value, *mf*, corresponding to its firing, the only output from the neuron. *S* represents the set of inputs to the neuron.

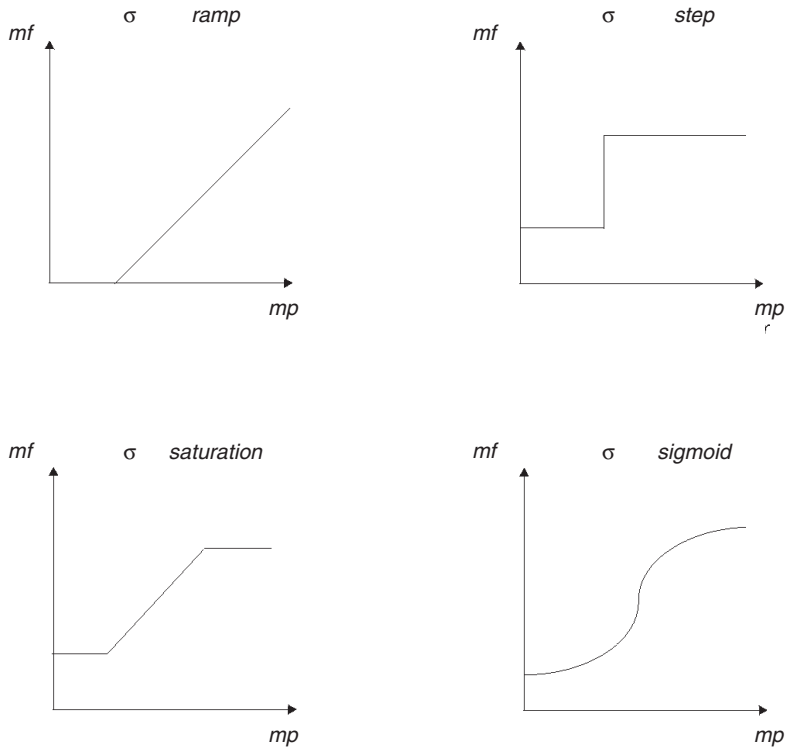

Figure 11 Common threshold functions.

work's outputs). The choice of transformation from S to *mp* defines the particular neural model utilized, including the dependence of *mp* on the neuron's previous history. The membrane potential *mp* is described by a simple first-order differential equation:

$$\tau \frac{dmp(t)}{dt} = f(S, mp, t)$$

depending on its input s. The choice of f defines the particular neural model utilized, including the dependence of *mp* on the neuron's previous history. For example, the *Leaky Integrator* model defines the membrane potential *mp* with the first-order differential equation

$$\tau \frac{dmp}{dt} = -mp + s$$

While neural networks are continuous in their nature, their simulated state is approximated by discrete time computations. For this reason, the NSL code for updating the membrane potentials of neurons will include the choice of an integration or approximation method to approximate the solution of differential equations.

The *average firing rate* or output of the neuron *mf* is obtained by applying some "threshold function" to the neuron's membrane potential:

$$mf = \sigma(mp)$$

where σ usually is described by a non-linear function. Basic threshold functions defined in NSL include *step*, *ramp*, *saturation* and *sigmoid*, whose behaviors are plotted in Fig. 11.

Synapses, the links among neurons, are—in biological systems—complex electrochemical systems and may be modeled in exquisite detail (see Chapter 2.3 on EONS). However, many models have succeeded with a very simple synaptic model, with each synapse carrying a connection weight that describes how neurons affect each other. The most common formula for the input to a neuron is given by:

$$sv_j = \sum_{i=0}^{n-1} w_{ji}\, uf_i$$

where uf_i is the firing of neuron u_i whose output is connected to the jth input line of neuron v_j, and w_{ji} is the weight for that link (*up* and *vp* are analogous to *mp*, while *uf* and *vf* are analogous to *mf*).

While module interconnections are specified in NSLM via an *nslConnect* method call, doing this with neurons would in general be prohibitively expensive considering that there may be thousands or millions of neurons in a single neural network. Instead, we use mathematical expressions similar to those used for their representation. Instead of describing neurons and links on a one-by-one basis, we extend the basic neuron abstraction into *neuron arrays* and *connection masks* describing spatial arrangements among homogeneous neurons and their connections, respectively. We consider uf_i the output from a single neuron in an array of neurons and sv_j the input

to a single neuron in another array of neurons. If mask w_{jk} (for $-d \leq k \leq d$) represents the synaptic weights from the uf_{j+k} (for $-d \leq k \leq d$) elements to sv_j, for each j, we then have:

$$sv_j = \sum_{k=-d}^{d} w_{jk}\, uf_{j+k}$$

where the same mask w is applied to the output of each neuron uf_{i+k} to obtain input sv_j. In NSLM, the equation is described by a single array *convolution* operation (operator "@"):

$$\mathrm{sv} = \mathrm{w@uf}$$

This kind of representation results in great conciseness, an important concern when working with a large number of interconnected neurons. Note that this is possible as long as connections are regular; otherwise, single neurons would still need to be connected separately on a one-by-one basis. This also suggests that the operation is best defined when the number of v and u neurons is the same, although a non-matching number of units can be processed using a more complex notation. There are special considerations with convolutions regarding edge effects—a mask centered on an element at the edge of the arrays extends beyond the edge of the array—depending on how out-of-bound array elements are treated. The most important alternatives are to treat edges as zero, wrap around array elements such as if the array was continuous at the edges, or replicate boundary array elements.

Simulation

Simulation involves interactively specifying aspects of the model that tend to change often, in particular parameter values and input patterns. Also, this process involves specifying simulation control and visualization aspects. For example, Fig. 12 shows five snapshots of the Buildup Cell activity after the simulation of one of the submodules in the *Superior Colliculus* of the *Crowley-Arbib* model shown in Fig. 1. We observe the activity of single neurons, classes of neurons, or outputs in response to different input patterns as the cortical command triggers a movement up and to the right. We see that the cortical command builds up a peak of activity on the Buildup Cell array. This peak moves towards the center of the array where it then disappears (this corresponds to the command for the eye moving towards the target, after which the command is no longer required).

Not only is it important to design a good model, but it is also important to design different kinds of graphical output to make clear how the model behaves. Additionally, an experiment may examine the effects of changing parameters in a model, just as much as changing the inputs. One of the reasons for studying the basal ganglia is to understand Parkinson's disease, in which the basal

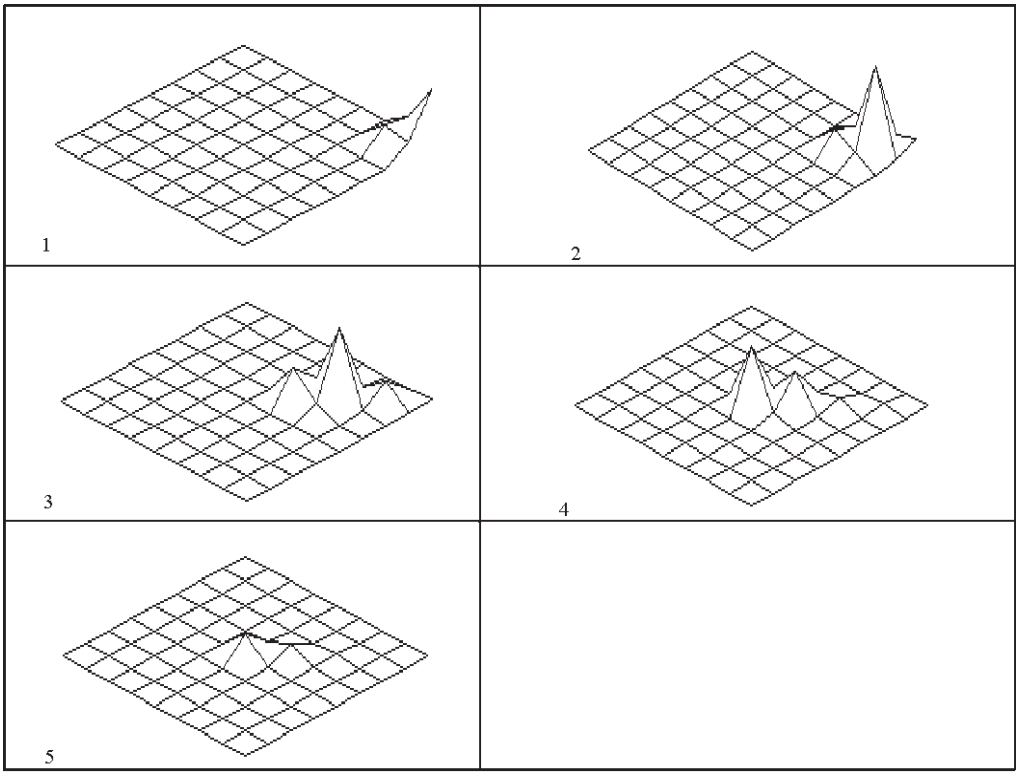

Figure 12 An example of Buildup Cell activity in the *Superior Colliculus* model of Fig. 1.

ganglia is depleted of a substance called dopamine, the depletion of which is a prime correlate of Parkinson's disease. The model of Fig. 2 (at a level of detail not shown) includes a parameter that represents the level of dopamine. Fig. 13 shows the response—saccade velocity, the behavior of a single cell, and the behavior of another single cell—to the brief presentation of a pair of targets. The "normal" model yields two saccades, one each in turn to the positions at which the two targets appeared; the "low-dopamine" model only shows a response to the first target, a result which gives insight into some of the motor disorders of Parkinson's disease patients.

Visualizing Model Architectures with the Schematic Capture System

There are two ways to develop a model architecture: by direct programming in NSLM or by using the *Schematic Capture System* (SCS), a visual programming interface to NSLM. SCS provides graphical tools to build hierarchical models. In SCS, the user graphically connects icons representing modules into what we call a *schematic*. Each icon can then be decomposed further into a schematic of its own. The benefit of having a schematic capture system is that modules can be stored in libraries and easily accessed by the schematic capture system. As more modules are added to the NSL model library, users will benefit by being able to create a rich variety of new models without having to write new code. The success of this will obviously depend on having good modules and documentation. When coming to view an existing model, the schematics make the relationship between modules much easier to visualize, besides simplifying the model creation process. As modules are summoned to the screen and interconnected, the system automatically generates the corresponding NSL module code.

Fig. 14 shows an example of a schematic. The complete schematic describes a single higher-level module, where rectangular boxes represent lower level modules or submodules. These modules can be newly defined modules or already existing ones. The lines describe connections among submodules, while small arrows describe input ports and straight lines describe output ports of modules. Pentagon-shaped boxes represent input (when lines come out from a vertex) and output (when lines

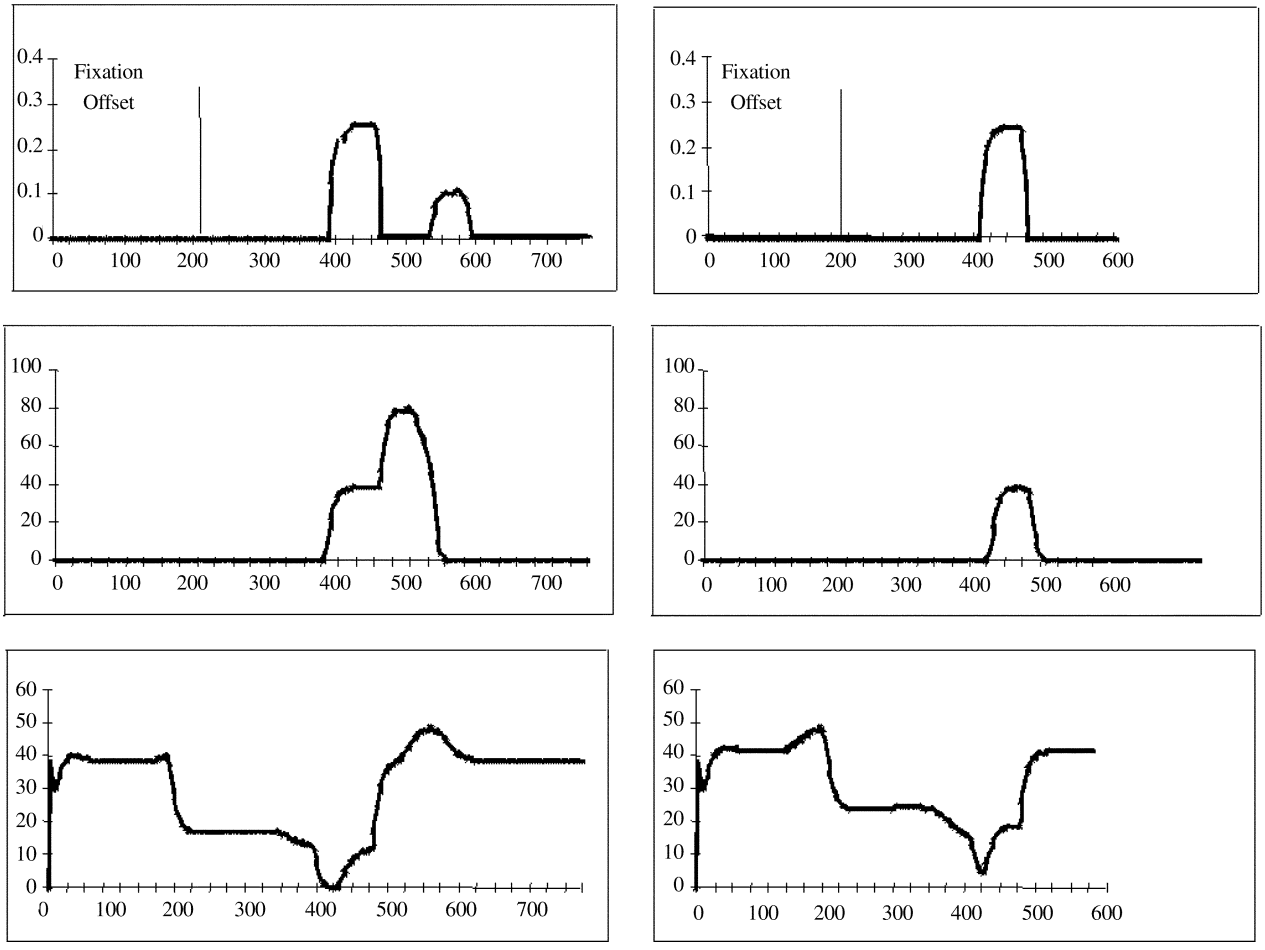

Figure 13 Cutting dopamine in half caused the second saccade not to be generated in response to a double saccade stimulus.

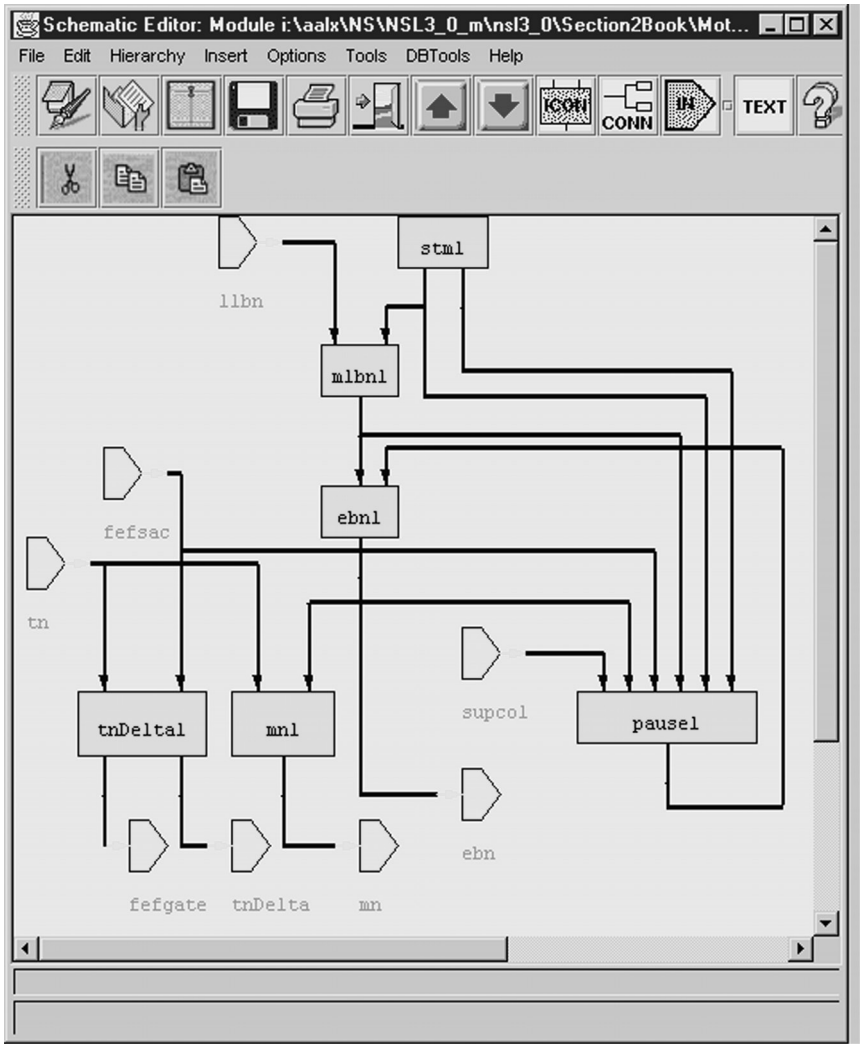

Figure 14 Schematic Editor showing the Motor Module for the *Dominey-Arbib* model. Pentagons in the shape of arrows show the input and output ports of the module.

point to a side) for the higher level module whose schematics are being described.

To select a model from those already existing in the NSL *Model Library*, we must first open the library by calling the Schematic Capture System. The system initially presents the *Schematic Editor* (SE) window. To execute an existing model, we select "Simulate Using Java" (or "Simulate Using C++") from the "Tools" menu. SCS then presents a list of available models. Once we choose the particular model, the system brings up the NSL Executive window presented in Fig. 15 together with an additional visualization window particular to this model, shown in Fig. 16. At this point we are ready to simulate the selected model. Yet, before we do that, we will quickly introduce the NSL Simulation Interface.

The NSL Executive window, shown in Fig. 15, is used to control the complete simulation process such as visualization of model behavior. Control is handled either via mouse-driven menu selections or by explicitly typing textual commands in the "nsls" shell. Because not all commands are available from the menus, the "nsls" shell is offered for more elaborate scripts.

The top part of the window (or header) contains the window name, NSL Executive, and the *Window Control* (right upper corner) used for iconizing, enlarging, and closing the window. Underneath the header immediately follows the *Executive Menu Bar*, containing the menus for controlling the different aspects involved in a simulation. The lower portion of the window contains the *Script window*, a scrollable window used for script command entry, recording, and editing. The Script Window is a superset of the pull-down menus in that any command that can be executed from one of the pull-down menus can also be typed in the Script window, while the opposite is not necessarily so. Furthermore, commands can also be stored in files and then loaded into the Script window at a later time. The Script window supports two levels of commands. The basic level allows *Tool Command Language* (TCL) commands (Ousterhout, 1994), while the second level allows NSLS

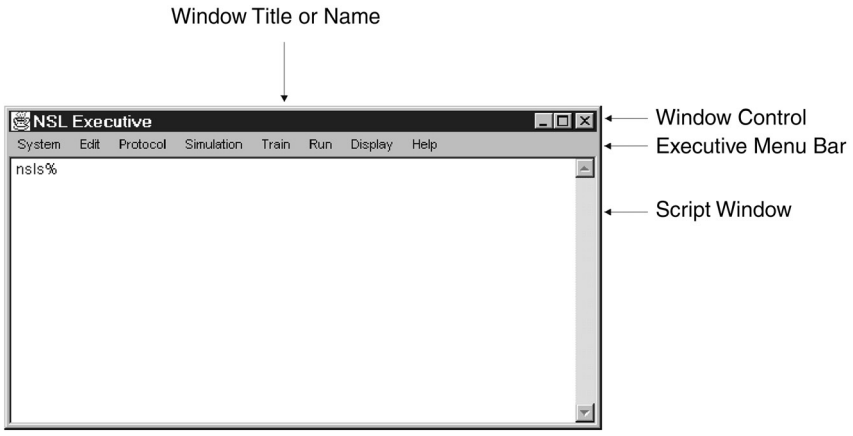

Figure 15 The NSL Executive window. The top part of the window contains the title and underneath the title is the Executive Menu Bar. The larger section of the window contains the Script Window.

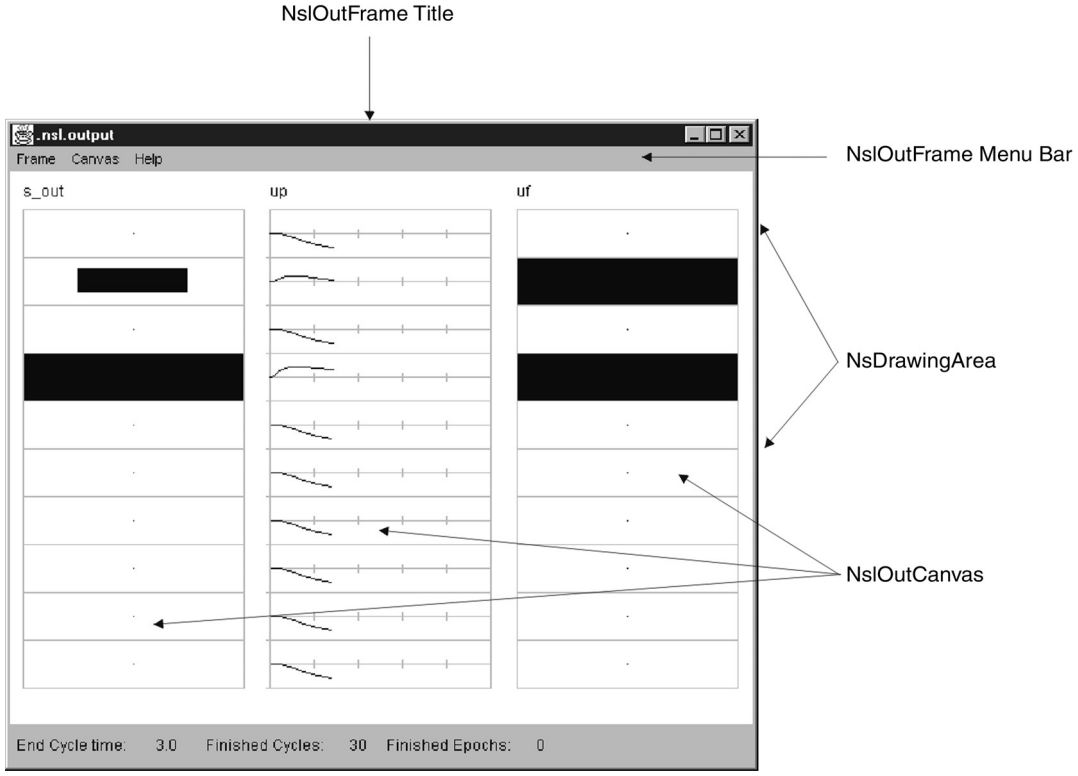

Figure 16 *MaxSelector* model NslOutFrame. The first canvas displays an area-level graph for the input variable s_out. The second canvas displays a temporal graph for the internal variable *up*, and the third canvas displays an area-level graph for the output variable *uf*.

commands. The NSLS commands have a special "nsl" prefix keyword to distinguish them from TCL commands. While there is a single NSL Executive window per simulation, there may be any number of additional output windows containing different displays. For example, the *Maximum Selector* model brings up the additional frame shown in Fig. 16.

The top part of the window contains the title or frame name and the very bottom of the frame contains the *Status line* or *Error message line*. When there are no errors or warnings, the status line displays the current simulation time. In the middle, the frame contains the *NslDrawingArea*. In this example, the drawing area contains three *NslOutCanvases*: the first and third

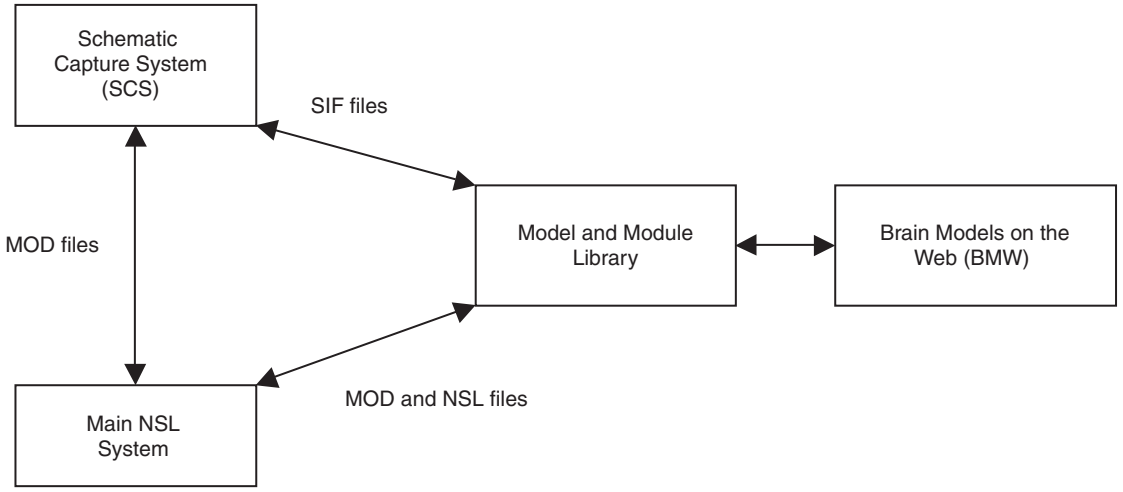

Figure 17 Schematic Capture System and its relation to the NSL system.

correspond to *Area* graphs while the second corresponds to a *Temporal* graph.

2.2.3 The NSL System

The complete NSL system is made of three components: the *Main System*, the *Schematic Capture System*, and the *Model/Module Library*, as shown in Fig. 17. Three file types are used as communication between the three components:

1. *MOD* files describing NSL models, executed by the Main System, stored in the Model Library, and optionally generated from SCS
2. *NSL* files describing NSL model simulation, executed by the Main System and stored in the Model Library
3. *SIF* files storing schematics information about the model stored in the Model Library as well.

The figure also shows BMW (Brain Models on the Web). This is not part of NSL but is a model repository under development by the USC Brain Project in which model assumptions and simulation results can be explicitly compared with the empirical data gathered by neuroscientists.

Main System

The Main NSL System contains several subsystems: the *Simulation subsystem*, where model interaction and processing takes place, and the *Window Interface subsystem*, where all graphics interaction takes place, as shown in Fig. 18. Note that we are now discussing the subsystems or modules that comprise the overall simulation system, not the modules of a specific neural model programmed with NSL. But, in either case, we take advantage of the methodology of object-oriented programming.

The Simulation subsystem is composed of:

1. *Control*, where external aspects of the simulation are controlled by the *Script Interpreter* and the Window Interface
2. *Scheduler*, where the model specified in *MOD file* gets executed
3. *Model Compiler*, where NSLM code is compiled and linked with NSL libraries to generate an executable file
4. *Script Interpreter*, where the simulation gets specified and controlled using NSLS, the script language

The Window Interface subsystem is composed of:

1. *Graphics Output*, consisting of all NSL graphic libraries for instantiating display frames, canvases, and graphs
2. *Graphics Input*, consisting of NSL window controllers to interact with the simulation and control its execution and parameters

Model/Module Library

NSL-developed models and modules are hierarchically organized in libraries. NSL supports two library structures. The first is called the *legacy* or *simple structure* as shown in Table 1. The second structure is a new one that the Schematic Capture System (SCS) builds and maintains. It is pictured in Table 2. The difference between the two is in how they manage modules. The legacy system gives a version number only to models, not modules. The SCS system gives version numbers to both models and modules and contains an extra directory, called the *exe* directory, for executables specific to different operating systems for the C++ version of the software.

There are several reasons for maintaining both systems. For the new system, we decided that it is better to

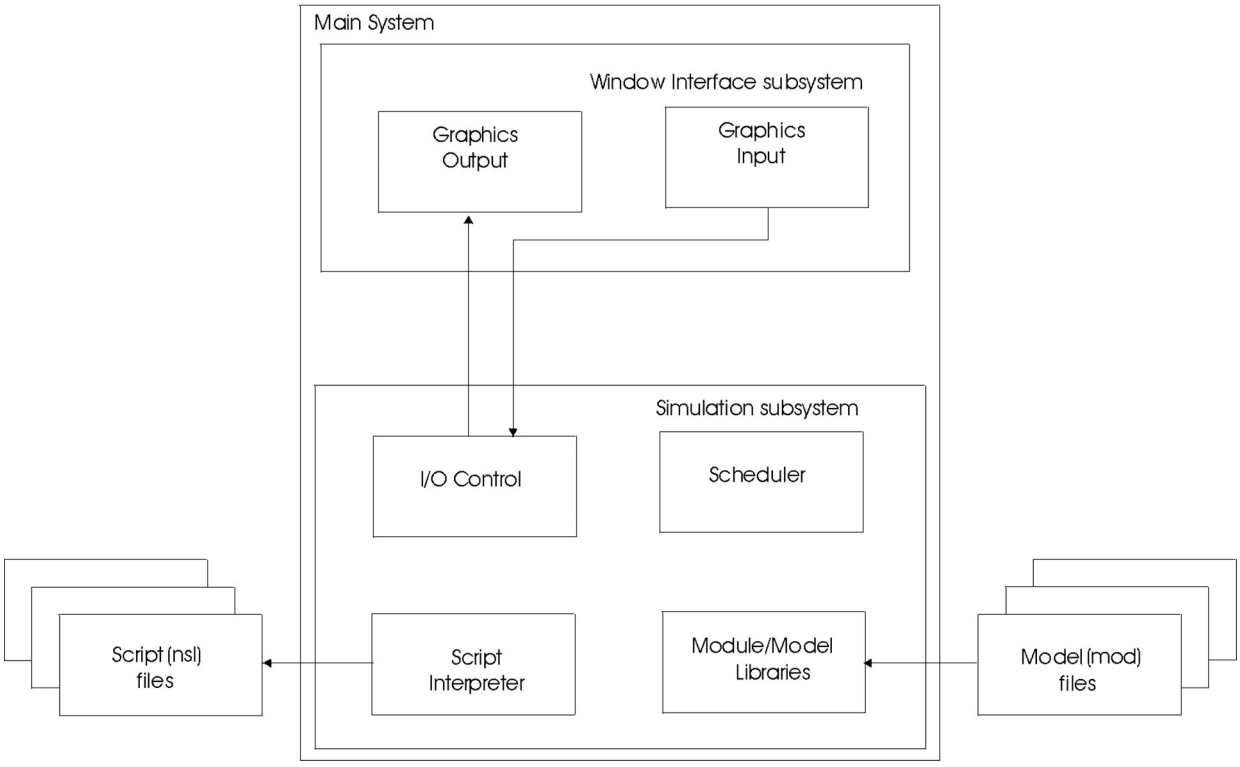

Figure 18 NSL's Main System composed of the Simulation and Window Interface subsystems

Table 1 NSL Model Library Hierarchy Organization for the Legacy System

nsl3_0		
Library Name		
Model Name		
Version Number		
io	src	doc

Table 2 NSL Model and Module Library Hierarchy Organization for the New System

nsl3_0			
Library Name			
Model or Module Name			
Version Number			
io	src	exe	doc

version (that is, create new versions of) the modules so that the user can try different versions of a module in his/her model and so that the modules can be shared more easily between models. Larger models will typically share modules and thus require the SCS management system. For the legacy/simple system, if you are not using SCS, it is easier to manage all of the module files in one directory, *src*. Also, if the modules are not intended to be shared or contributed to Brain Models on the Web (BMW), then they do not necessarily need to be versioned.

The general organization of legacy architecture is shown in Table 1, where the levels correspond to directories. The root of the hierarchy is "nsl3_0" the current system version. A library name is defined to distinguish between libraries. Obviously, there may be multiple model libraries. Each library may contain multiple *models* identified by their corresponding name. Each model is then associated with different implementations, each identified by its corresponding *version number* (version numbers start at 1_1_1). The bottom level of the directory hierarchy contains the directories where the actual model or module files are stored. The *io* directory stores input and output files, usually in the form of NSLS script files. The *src* directory contains source code that must be compiled and is written in the NSLM modeling language; this directory also includes files produced from the compilation including executables. The *doc* directory contains any documentation relevant to the model, including theoretical background, why certain values were chosen for certain parameters, what is special about each of the protocols, how to perform more sophisticated experiments, relevant papers, etc. All models given in the NSL book (Weitzenfeld *et al.*, 2000) were originally developed using the legacy system; thus, as of early 2000, this is what will be found at our Website: http://www-hbp.usc.edu/Projects/nsl.htm.

SCS allows the user to create new libraries as well as add new revisions to existing models and modules. The user can browse and search the libraries for particular

models or modules. When building a schematic, the user has the choice of choosing the most recent modification of a model or module or sticking with a fixed version of that model or module. If the user chooses a specific version, this is called "using the *fixed* version." If the user specifies "0_0_0" then the most current version of the module would be used instead. This is called "using the *floating* version." Each individual library file stores *meta-data* describing the software used to create the corresponding model/module. Because SCS generates and manages the NSLM source code files, it also provides a utility for generating the compilation scripts necessary to build the model

2.2.4 Simulating a Model – The Maximum Selector Model

If a model is a discrete-event or discrete-time model, the model equations explicitly describe how to go from the state and input of the network at time t to the state and output after the event following t is completed or at time $t + 1$ on the discrete time scale, respectively. However, if the model is continuous-time, described by differential equations, then simulation of the model requires that we must replace the differential equation by some discrete-time numerical method (e.g., Euler or Runge-Kutta) and choose a simulation time step (t so the computer can go from state and input at time t to an *approximation* of the state and output at time $t + \Delta t$. In each case, the simulation of the system proceeds in steps, where each *simulation cycle* provides the updating for one time step of the chosen type. Simulation involves the following aspects of model interaction: (1) simulation control, (2) visualization, (3) input assignment, and (4) parameter assignment.

Simulation Control

Simulation control involves the execution of a model. The Executive window's "Simulation," "Train," and "Run" menus contain options for starting, stopping, continuing, and ending a simulation during its training and running phase, respectively.

Visualization

Model behavior is visualized via a number of graphics displays. These displays are drawn on canvases, *NslOutCanvas*, each belonging to a *NslOutFrame* output frame. Each NslOutFrame represents an independent window on the screen containing any number of NslOutCanvases for interactively plotting neural behavior or variables in general. NSL canvases can display many different graph types that display NSL numeric objects—objects containing numeric arrays of varying dimensions. For example the *Area* graph shown in Fig. 16 displays the activity of a one-dimensional object at every simulation cycle. The size of the dark rectangle represents a corresponding activity level. On the other hand, the *Temporal* graph shown displays the activity of a one-dimensional objects as a function of time (in other words, it keeps a history).

Input Assignment

NSL supports input via custom-designed input windows as well as by the default user interface using script commands in the NSLS language using the *Script window*. The Script window also allows one to save and load script files.

Parameter Assignment

Simulation and model parameters can be interactively assigned by the user. Simulation parameters can be modified via the "Options" menu while model parameters are modified via the Script window. Additionally, some models may have their own custom-designed window interfaces for parameter modification.

2.2.5 Maximum Selector Model

The remaining sections of this chapter illustrate model simulation using the *Maximum Selector* model. It is a very simple model but allows us to briefly present a number of key features of the NSL environment. The models described in BMW offer far fuller use of the NSL system, or of related approaches to modular architecture of large-scale neural systems. To get one of the models, just visit our Website at http://www-hbp.usc.edu/Projects/nsl.htm. From that page, a list of models executable from NSL will be displayed, and you can pick and choose which one you wish to download.

The *Maximum Selector* neural model (Amari and Arbib, 1977) is an example of a biologically inspired neural network and is shown in Fig. 19. The network is based on the *Didday* model for prey selection (Didday, 1976) and is more generally known as a *Winner Take All* (WTA) neural network. The model uses competition mechanisms to obtain, in many cases, a single winner in the network where the input signal with the greatest strength is propagated along to the output of the network. External input to the network is represented by s_i (for $0 \leq i \leq n–1$). The input is fed into neuron u, with up_i representing the membrane potential of neuron u, while uf_i represents its output. uf_i is fed into neuron v as well as back into its own neuron. vp represents the membrane potential of neuron v, which plays the role of inhibitor in the network. w_m, w_{ui}, and w_i represent

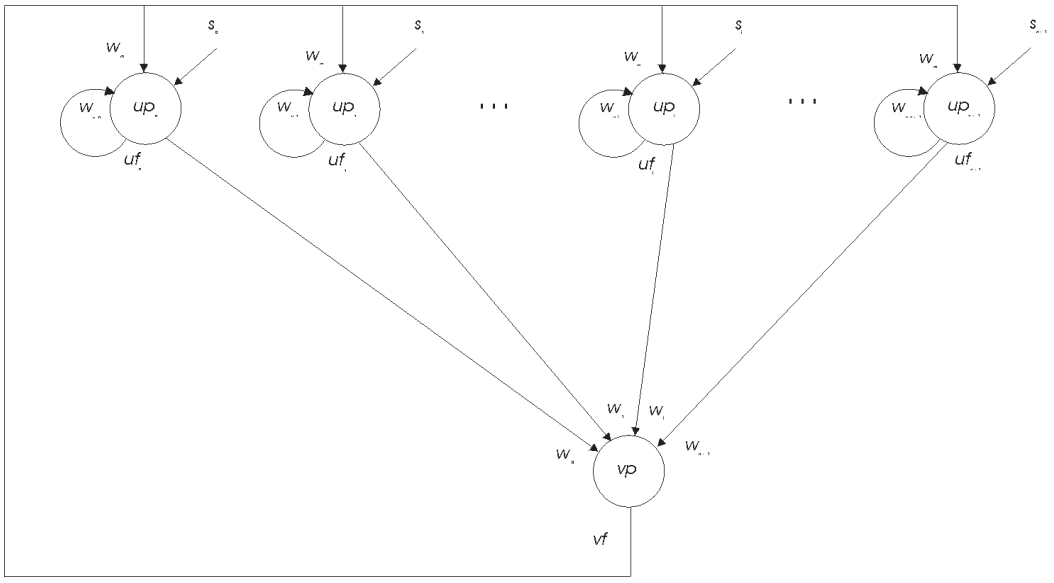

Figure 19 The neural network architecture for the Maximum Selector (Didday, 1976; Amari and Arbib, 1977)[AU8] where s_i represents input to the network and up_i and *vp* represent membrane potentials, while uf_i and *vf* represent firing rates; w_m, w_{ui}, and w_i correspond to connection weights.

connection weights, whose values are not necessarily equal.

The neural network is described by the following set of equations:

$$\tau_u \frac{du_i(t)}{dt} = -u_i + w_u f(u_i) - w_m g(v) - h_1 + s_i \qquad (2.1)$$

$$\tau_v \frac{dv}{dt} = -v + w_n \sum_{i=1}^{n} f(u_i) - h_2$$

where w_u is the self-connection weight for each u_i, w_m is the weight for each u_i for feedback from v, and each input s_i acts with unit weight; w_n is the weight for input from each u_i to v. The threshold functions involve a *step* for (fu_i):

$$f(u_i) = \begin{cases} 1 & u_i > 0 \\ 0 & u_j \leq 0 \end{cases} \qquad (2.2)$$

and a *ramp* for $g(v)$:

$$g(v) = \begin{cases} v & v > 0 \\ 0 & v \leq 0 \end{cases}$$

Again, the range of i is $0 \leq i \leq n$–1, where n corresponds to the number of neurons in the neural array u. Note that the actual simulation will use some numerical method to transform each differential equation of the form $\tau\, dm/dt = f(m,s,t)$ into some approximating difference equation of the form $m(t + t) = F(m(t), s(t), t)$ which transforms state $m(t)$ and input $s(t)$ at time t into the state $m(t + \Delta t)$ of the neuron one "simulation time step" later.

As the model equations get repeatedly executed, with the right parameter values, u_i values receive positive input from both their corresponding external input and local feedback. At the same time negative feedback is received from v. Because the strength of the negative feedback corresponds to the summation of all neuron output, as execution proceeds only the strongest activity will be preserved, resulting in many cases in a "single winner" in the network.

To execute the simulation, having chosen a differential equation solver and a simulation time step (or having accepted the default values), the user would simply select "Run" from the NSL Executive's Run menu, as shown in Fig. 20. We abbreviate this as Run → Run.

The output of the simulation would be that as shown in Fig. 21.

The resulting statistics on how long the simulation took to run are displayed in the Executive window's shell, as shown in Fig. 22.

Recall that NSLS is the NSL scripting language in which one may write a script file specifying, for example, how to run the model and graph the results. The user may thus choose to create a new script, or retrieve an existing one. In the present example, the user gets the system to load the NSLS script file containing preset graphics, parameters, input, and simulation time steps by selecting "System → Nsls file …", as shown in Fig. 23.

From the file selection pop-up window we first choose the "nsl" directory and then *MaxSelectorModel*, as shown in Fig. 24. Alternatively, the commands found in the file could have been written directly into the Script window, but it is more convenient the previous way.

Simulation control involves setting the duration of the model execution cycle (also known as the delta-t or simulation time step). In all of the models we will present, we will preset the simulation control parameters within the model; however, to override these settings the user

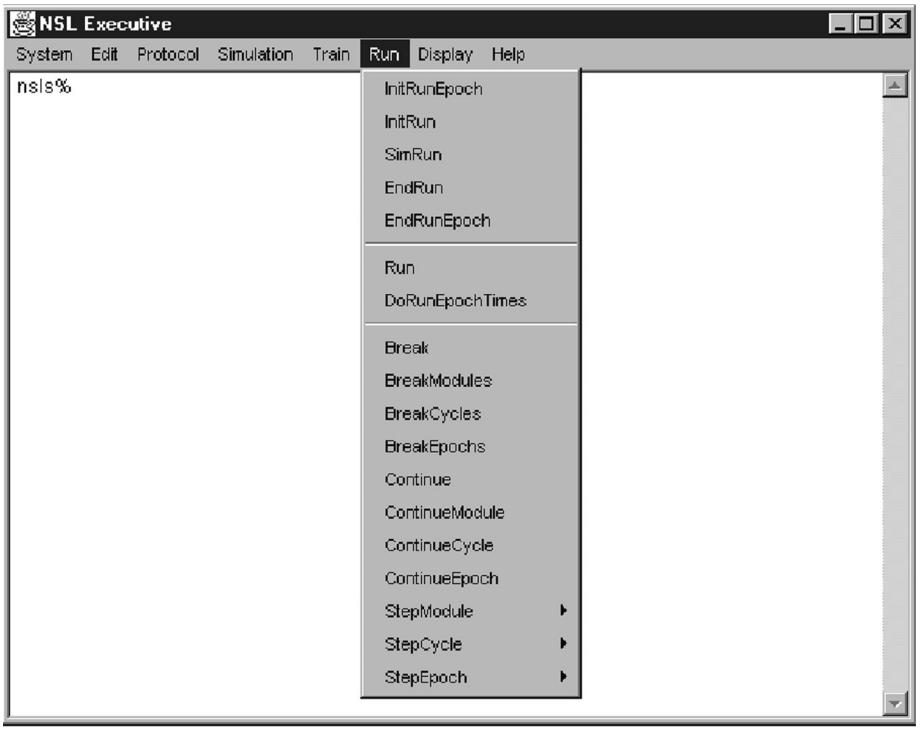

Figure 20 The "Run → Run" menu command.

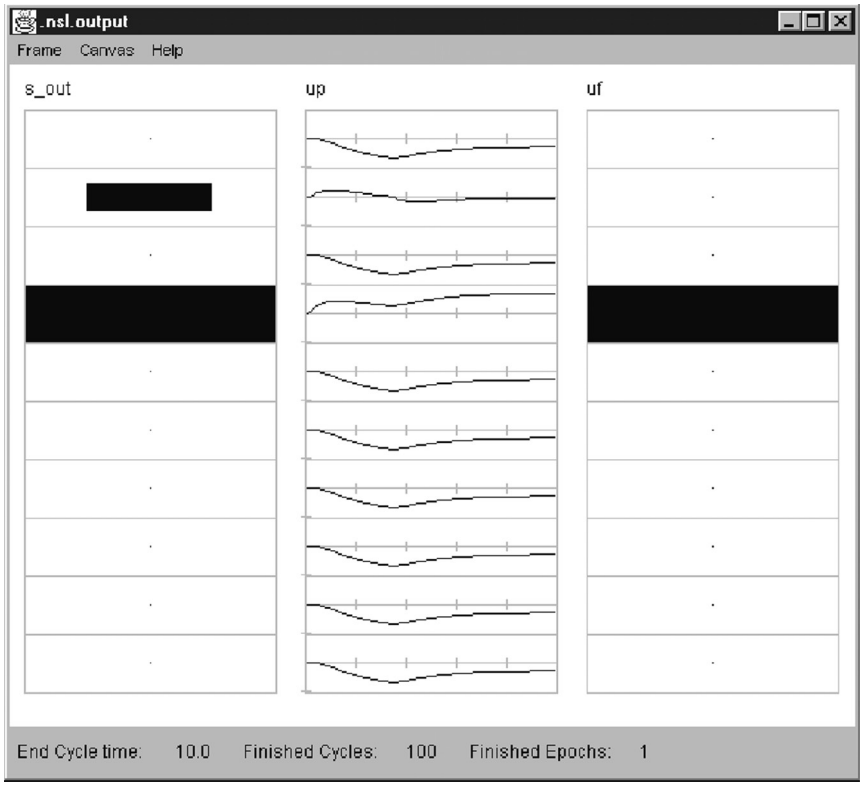

Figure 21 Output of the *MaxSelector* model. Notice that the second and fourth elements in the "maxselector.u1.up" membrane potential layer are affected by the input stimuli; however, the "winner take all" circuit causes the fourth element to dominate the output, as seen in the firing rate, "maxselector.u1.uf."

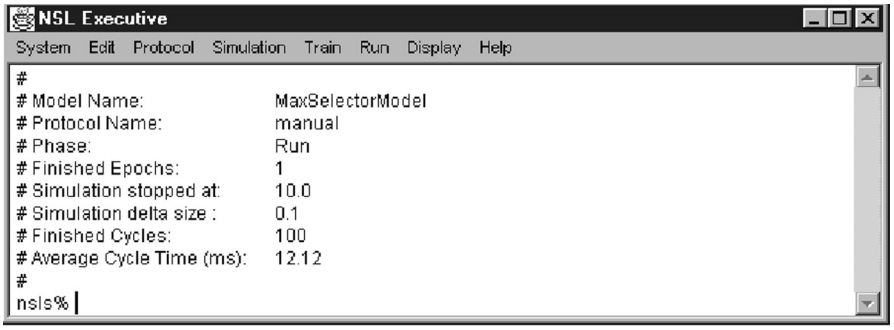

Figure 22 Executive window output from *Maximum Selector* model run.

Figure 23 Loading a "nsls" script file into the Executive.

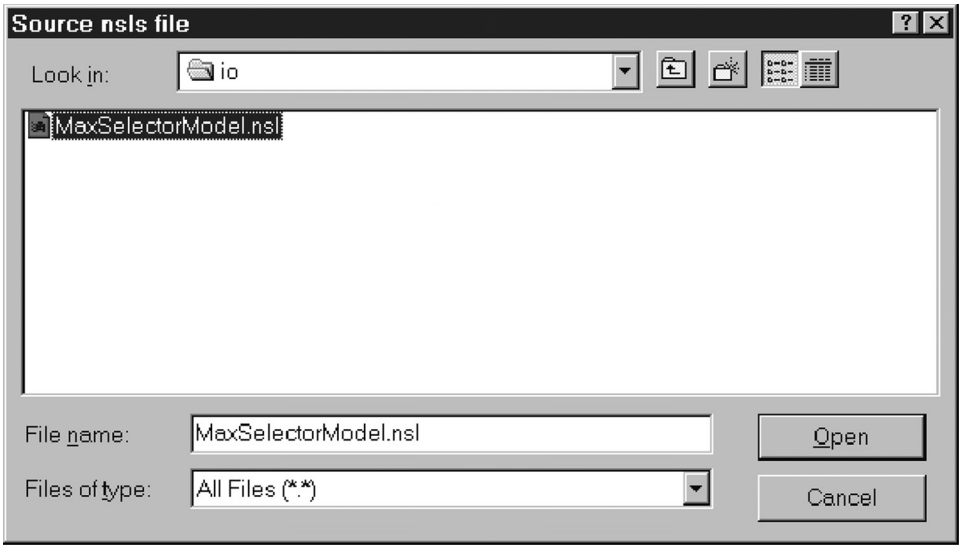

Figure 24 The MaxSelectorModel script loaded into the Executive.

can select System → Set → RunEndTime and System → Set → RunDelta, as shown in Fig. 25.

A pop-up window appears showing the current parameter value that may be modified by the user. In this model we have set the *RunEndTime* to 10.0, as shown in Fig. 26, and *RunDelta* to 0.1 giving a total of 100 execution iterations. These values are long enough for the model to stabilize on a solution.

To execute the actual simulation, we select "Run" from the "Run" menu, as we did in Fig. 20. The user may stop the simulation at any time by selecting the "Run" menu and then selecting "break." We abbreviate this as Run → Break. To resume the simulation from the interrupt point, select Run → Continue.

The model output at the end of the simulation is shown in Fig. 21. The display shows input array *S_out*

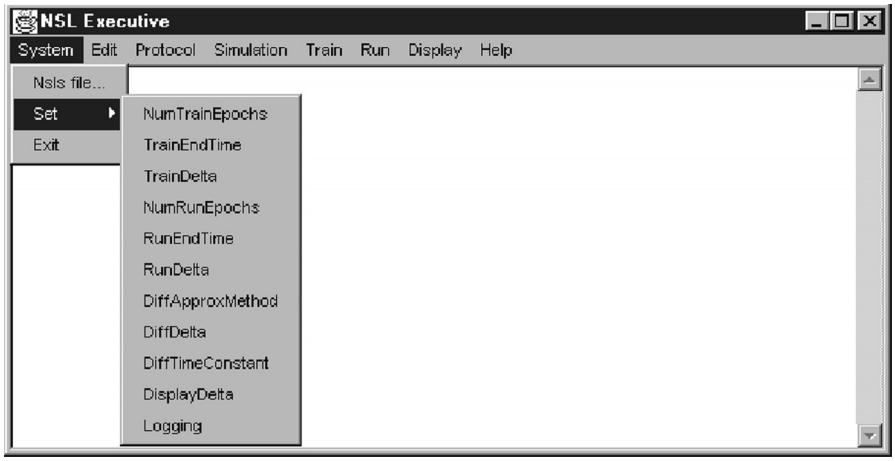

Figure 25 Setting system control parameters.

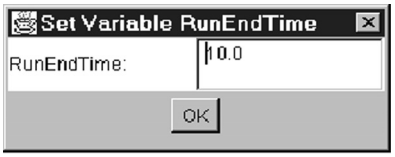

Figure 26 RunEndTime parameter setting.

with an Area type graph (i.e., the area of the black rectangle codes the size of the corresponding input), while array *up*, with a Temporal type graph, shows the time course for *up*. The last canvas shows another Area type graph for *uf* at the end of the simulation. The largest input in *S_out* determines the only element of *S_out* whose activity is still positive at the end of the simulation as seen in *uf*—the network indeed acts as a maximum selector.

The *Maximum Selector* model example is quite simple in that the input *S_out* is constant. In the example chosen, it consists of only two different positive values (set to 0.5 and 1.0) while the rest are set to zero (total of 10 elements in the vector). In general, input varies with time. Because input is constant in the present case, it may be set similarly to any model parameter. To assign values to parameters, we use the "nsl set" command followed by the variable and value involved. If we are dealing with dynamic input, we have different alternatives for setting input. One is to specify a "nsl set" command with appropriate input values every time input changes. Another alternative is to specify the input directly inside the model description or through a custom interface.

Parameters whose values were not originally assigned in the model description, or that we may want to modify, are specified interactively. Two parameters of special interest in the model are the two thresholds, *hu* and *hv*. These two parameters are restricted as follows: $0 \leq hu$, and $0 \leq hv < 1$. (For the theory behind these restrictions, see Arbib, 1989, Sec.4.4.). Their initial values are set to 0.1 and 0.5, respectively. To exercise this model, the user can change both the input and parameter values to see different responses from the model. We suggest trying different combinations of input values, specifying different number of array elements receiving these values. In terms of parameters, we suggest changing values for *hu* and *hv*, including setting them beyond the mentioned restrictions. Every time parameters or input changes, the model should be reinitialized and executed by selecting the "run" menu option.

2.2.6 Available Resources

As mentioned in this chapter, the USC Brain Project Website contains the links to the NSL, SCS, and BMW home pages. The NSL software, documentation, and an exemplary set of models written in NSL can be found from the page: http://www-hbp.usc.edu/Projects/nsl.htm. The SCS software, documentation, and models can be found from the main NSL home page. The models that have been formatted and annotated for BMW can be found from http://www-hbp.usc.edu/Projects/bmw.htm. In addition, the USCBP Website contains a general modeling page that contains some models that are neither in the NSL flat file repository nor the BMW repository. These models can be found from the page: http://www-hbp.usc.edu/Thrusts/modeling.htm.

Acknowledgment

This research was supported in part by P20 Program Project grant 5–P20–52194 from the Human Brain Project (NIMH, NIDA, and NASA) and in part by a grant from NSF-CONACyT. This chapter is based on material from Weitzenfeld *et al.* (2001).

References

Arbib, M. A. (1992). Schema theory, in *The Encyclopedia of Artificial Intelligence*. 2nd ed. (S. Shapiro, Ed.), pp. 1427–1443, Wiley-Interscience, New York.

Bower, J. M., and Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation Systems* 2nd ed.. TELOS/Springer-Verlag, Berlin/New York.

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Computation* **9**, 1179–1209.

Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol. London*, **117**, 500–544.

Rall, W. (1959). Branching dendritic trees and motoneuron membrane resistivity. *Exp. Neurol.* **2**, 503–532.

Weitzenfeld, A. (1992). A Unified Computational Model for Schemas and Neural Networks in Concurrent Object-Oriented Programming, Ph.D. thesis, Computer Science, University of Southern California, Los Angeles.

Weitzenfeld, A., and Arbib, M. A. (1991). A concurrent object-oriented framework for the simulation of neural networks, in Proceedings of ECOOP/OOPSLA 1990 Workshop on Object-Based Concurrent Programming. *SIGPLAN, OOPS Messenger* **2(2)**, 120–124.

Weitzenfeld, A., Alexander, A. and Arbib. M. A. (2001). *The NSL Neural Simulation Language*. The MIT Press, Cambridge, MA.

Yonezawa, A., and M. Tokoro, Eds. (1987). *Object-Oriented Concurrent Programming*. The MIT Press, Cambridge, MA.