1 Introduction

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, extended to provide modularity at the application level as well. In this chapter we discuss these basic concepts and how NSL takes advantage of them.

1.1 Neural Networks

Neural network simulation is an important research and development area extending from biological studies to artificial applications. Biological neural networks are designed to model the brain in a faithful way while artificial neural networks are designed to take advantage of various “semi-neural” computing techniques, especially the use of different learning algorithms in distributed networks, in various technological domains. Challenges vary depending on the respective areas although common basic tasks are involved when working with neural networks: *modeling* and *simulation*.

Modeling

Modeling or development of a neural network or neural architecture depends on the type of network being constructed. In the case of artificial neural modeling, neural architectures are created to solve the application problem at hand, while in the case of biological modeling neural architectures are specified to reproduce anatomical and physiological experimental data. Both types of network development involve 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.

For biological modeling, the neuron model varies depending on the details being described. Neuron models can be very sophisticated biophysical models, such as compartmental models (Rall 1959) in turn based on the Hodgkin-Huxley model (Hodgkin and Huxley 1952). 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. And sometimes even simpler neural models are enough, in particular for artificial networks, as with discrete binary models where the neuron is either on or off at each time step, as in the McCulloch-Pitts model (McCulloch and Pitts 1943).

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, artificial neural networks—as do many biological models—involve learning, requiring an additional training phase in the model architecture.

To generate a neural architecture the network developer requires a modeling language sufficiently expressive to support their representation. On the other hand, the language should be extensible enough to integrate with other software systems, such as to obtain or send data. In general, a neural network modeling or development environment should support a set of basic structures and functions to simplify the task of building new models as well as interacting with them.

Clearly, the user’s background plays an important role in the sophistication of the development environment. Novice users depend almost completely on the interactivity provided through window interfaces, while more sophisticated users usually desire extensibility in the form of programming languages.

Simulation

Simulation of neural network architectures also varies depending on whether it relates to artificial or biological networks. Artificial neural networks particularly those involving learning usually require a two-stage simulation process: an initial training phase and a subsequent processing or running phase. Biological networks usually require a single running phase (in which behavior and learning may be intertwined).

Simulation 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 then involves analyzing the results, both visual and numerical, generated by the simulation; 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 or may require more structural modifications at the neural architecture level going back to the development phase. Otherwise the model is simulated again with newly specified input. Simulation also involves selecting one of the many approximation methods used to solve neural dynamics specified through differential equations.

In addition, the environment requirements can change when moving a model from development phase to 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. Parallelism and distributed computing will increasingly play key roles in speeding up computation.

1.2 Modularity, Object-Oriented Programming, and Concurrency

Modularity, object-oriented programming and concurrency play an important part in building neural networks in NSL as well as in their execution. Furthermore, the actual NSL system is built based on object-oriented technology.

Modularity in Neural Networks

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 hard 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, figure 1.1 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.

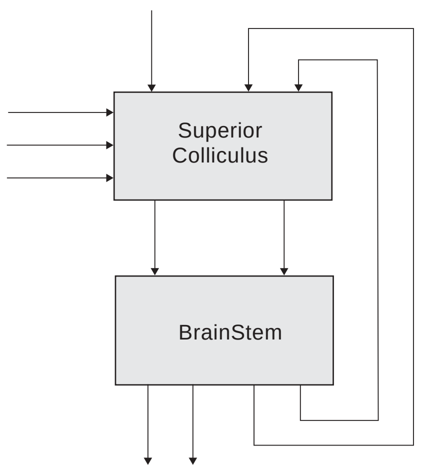

Figure 1.1
The diagram above shows two interconnected modules, the Superior Colliculus (SC) and the Brainstem. Each module is decomposed into several submodules (not shown here) each imple-mented as an array of neurons identified by their different physiological response when a monkey makes rapid eye movements.

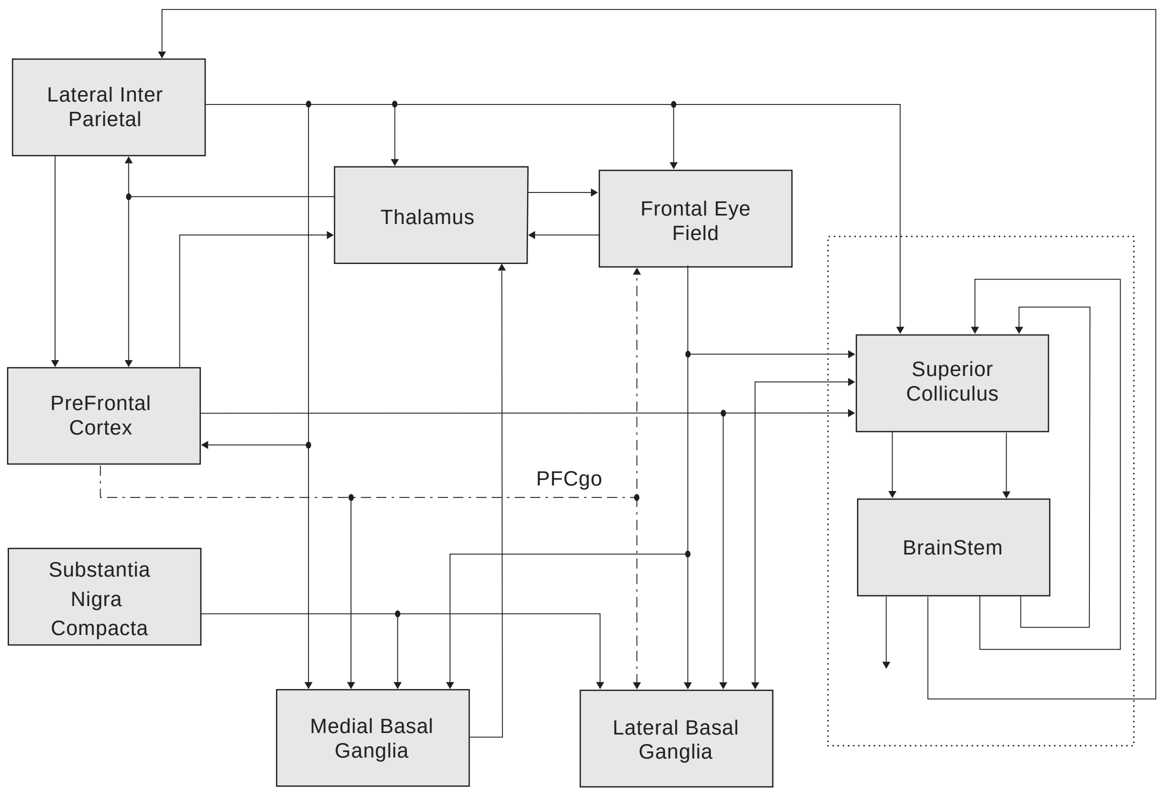

Figure 1.2
The diagram shows the SC and BrainStem modules from figure 1.1 embedded in a much larger model of interacting brain regions.

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, figure 1.2 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. For example, figure 1.3 shows how the single *Prefrontal Cortex* module (*PFC*) can be further broken down into four submodules, each a crucial brain region involved in the control of movement.

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.

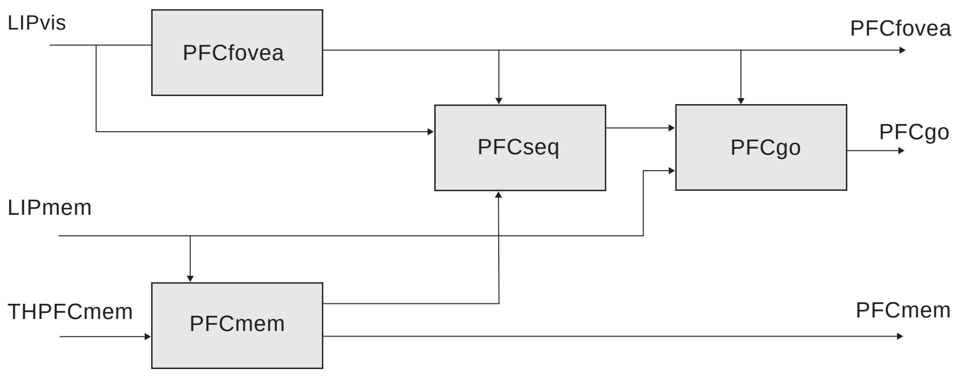

Figure 1.3
The *Prefrontal Cortex* (PFC) model is further decomposed into 4 submodules.

Object-Oriented Programming

Object-oriented technology has existed for more than thirty 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. Since data and functions are global to a program 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 hard 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. Since 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 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 is only part of writing better software as well as neural models. 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. The NSL software supplied with this book is 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. We will discuss this more in the Future Directions chapter.

1.3 Modeling and Simulation in NSL

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

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. This complexity is only viewed at the bottom-level corresponding to the actual neural networks. 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.

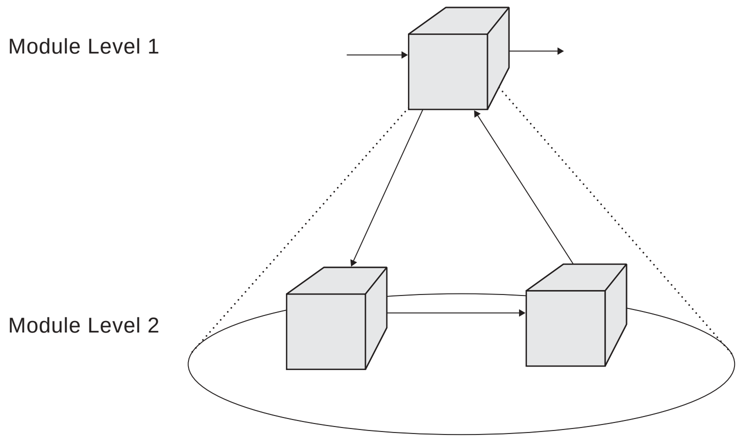

Figure 1.4
The NSL computational model is based on hierarchi-cal modules. A module at a higher level (level 1) is decomposed into submod-ules (level 2). These sub-modules are themselves modules that may be further decomposed. Arrows show data communication among modules.

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 figure 1.4.

Thus a given module may either be decomposed into a set of smaller modules or maybe a “leaf module” that may be implemented in different ways, where neural networks are of particular interest here. The hierarchical module decomposition 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.

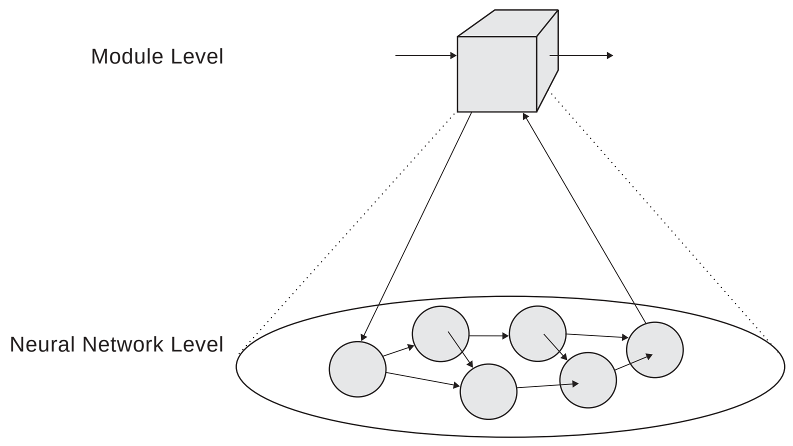

Figure 1.5
A module in NSL implemented by neural networks made of multiple neurons.

Neural Networks
Some modules will be implemented as neural networks where every neuron becomes an element or attribute of a module, as shown in figure 1.5. (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 1997) systems were designed specifically to support the type of 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, this book will focus on the simulation of large-scale properties of neural networks modeled with relatively simple neurons.

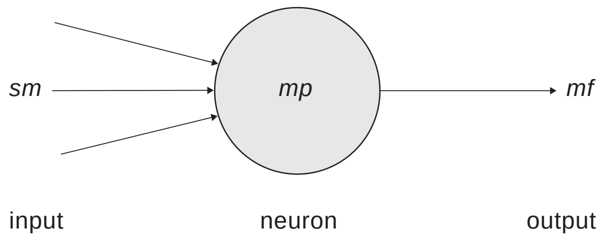

Figure 1.6
Single compartment neural model represented by a value *mp* corresponding to its membrane potential, and a value *mf* corresponding to its firing, the only output from the neuron. *sm* represents the set of inputs to the neuron.

We consider the neuron shown in figure 1.6 to be "simple" since its internal state is described by a single scalar quantity, membrane potential *mp*, its input is *sm* 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 network's outputs). The choice of transformation from *sm* 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,

$$\tau \frac{dmp(t)}{dt} = f(sm, mp, t) \tag{1.1}$$

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. In this example we present the leaky integrator. The *leaky integrator* model is described by

$$f(sm, mp, t) = -mp(t) + sm(t) \tag{1.2}$$

while the *average firing rate* or output of the neuron, *mf*, is obtained by applying some "activation function" to the neuron's membrane potential,

$$mf(t) = \sigma(mp(t)) \tag{1.3}$$

where σ usually is described by a non-linear function also known as *threshold* functions such as *ramp*, *step*, *saturation* or *sigmoid*. The general idea is that the higher the neuron membrane potential, the higher the firing rate, and thus the greater its effect on other neurons to which it provides input.

The neural network itself is made of any number of interconnected neurons where the most common formula for the input sm_j to a neuron m_j from the output of a neuron m_i as shown in figure 1.7 is given by,

$$sv_j = \sum_{i=0}^{n-1} w_{ij} uf_i \tag{1.4}$$

where $uf_i(t)$ is the firing rate of the neuron whose output is connected to the *i*th input line of neuron v_j, and w_{ij} is the corresponding weight on that connection (*up* and *vp* are analogous to *mp*, while *uf* and *vf* are analogous to *mf*). These interconnections are called *excitatory* or *inhibitory* depending on whether the weight w_{ij} is positive or negative.

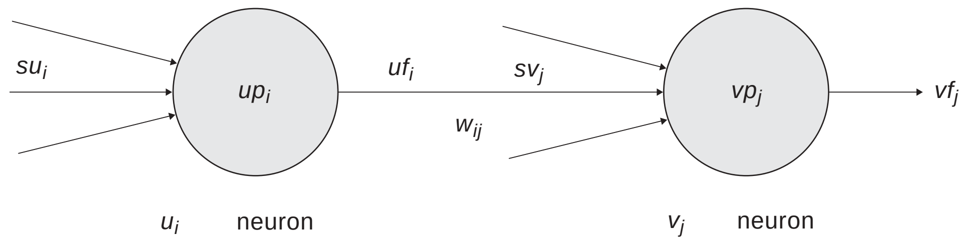

Figure 1.7 Interconnection between two neurons showing the input sv_j to a neuron v_j from the output of a neuron u_i with connection weight w_{ij}

When modeling a large number of neurons it becomes extremely tedious to individually name each one of the neurons. In the brain as well as in many neural engineering applications, we often find neural networks structured into two-dimensional arrays, with regular connection patterns between various arrays. For this reason, as part of our modeling primitives, we extend a simple single neuron into neuron arrays and single neuron-to-neuron links into connection masks, describing spatial arrangements among homogeneous neurons and their connections, respectively. If mask w_k (for $-d \le k \le d$) represents the synaptic weight from the uf_{j+k} (for $-d \le k \le d$) elements to v_j element for each *j*, we then have

$$sv_j = \sum_{k=-d}^{d} w_k uf_{j+k} \tag{1.5}$$

The computational advantage of introducing such concepts when describing a "regular" neural network, as shall be seen in chapter 3, is that neuron arrays and interconnection masks can then be more concisely represented. Interconnections among neurons would then be processed by a spatial convolution between a mask and an array. Once interconnections are specified between neurons or neural arrays, we only need to specify network input; weights and any additional parameter before simulation can take place.

Simulation

The simulation process starts with a model already developed. 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, figure 1.8 shows five snapshots of the Buildup Cell activity after the simulation of one of the submodules in the Superior Colliculus of the Crowley and Arbib model shown in figure 1.1. 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).

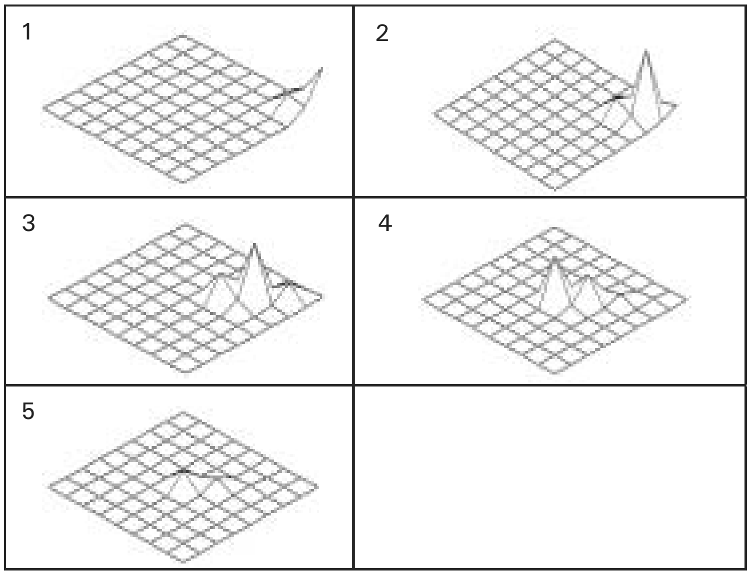

Figure 1.8
An example of Buildup Cell activity in the Superior Colliculus model of figure 1.1.

It is not only important to design a good model, 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 ganglia are depleted of a substance called dopamine, whose depletion is a prime correlate of Parkinson's disease. The model of figure 1.2 (at a level of detail not shown) includes a parameter that represents the level of dopamine. 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. The actual model is described in detail in chapter 15. We shall describe the simulation process in more detail in chapter 2.

1.4 The NSL System

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 USC 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. 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 Alfredo Weitzenfeld's team), again with Arbib involved in the overall design. Both environments support similar modeling and simulation, where each one offers different advantages to the user.

The advantages with Java are

- *portability*: Code written in Java runs without changes "everywhere";
- *maintainability*: Java code requires maintaining one single software version for different operating systems, compilers and other software on different platforms.
- *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.

The advantages with C++ are

- *efficiency*: Since C++ is an extension to C, C++ models get simulated on top of one of the most efficient execution languages;
- *integration*: C++ code may be directly integrated with a large number of software packages already in existence written in C++;
- *linkage to hardware*: Currently most linkages to robots are done through C and C++; however, more and more of these systems are moving to Java.

The great advantage on having support for both environments is the ability to switch between the two of them to get the best of each world with minimum effort.

The complete NSL system is made of three components: the **Simulation System**, the **Schematic Capture System** and the **Model/Module Libraries**, as shown in figure 1.9. Three file types are used as communication between the three modules:

- **mod** files describing NSL models, executed by the Simulation System, stored in the Model Library and optionally generated from SCS,
- **nsl** files describing NSL model simulation, executed by the Simulation System and stored in the Model/Module Libraries,
- **sif** files storing schematic information about the model stored in the Model/Module Libraries as well.

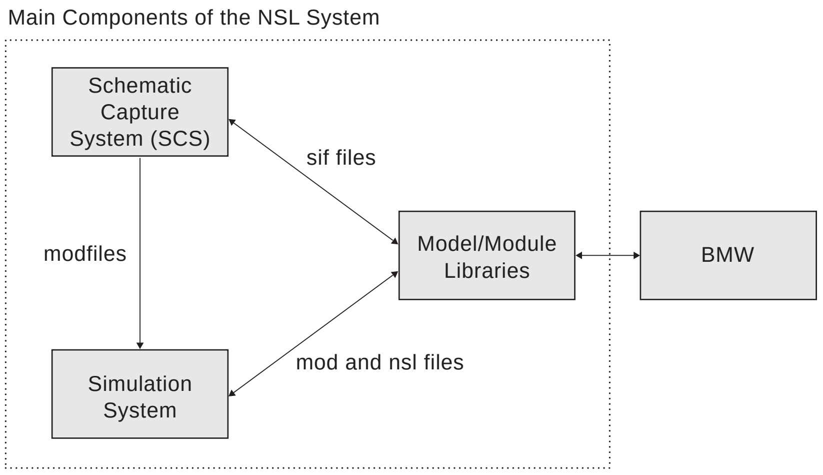

Figure 1.9
Schematic Capture System and its relation to the NSL System.[1]

Simulation System

The NSL Simulation System comprises 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 figure 1.10. 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 subsystems of the **Simulation System** are:

- I/O **Control** where external aspects of the simulation are controlled by the *Script Interpreter* and the Window Interface;
- **Scheduler** which executes the model and modules in a specific sequence.
- **Model Compiler** where NSLM code is compiled and linked with NSL libraries to generate an executable file;
- **Script Interpreter** that can be used to specify parameters and to control the simulation.
- The subsystems of the **Window Interface** are:
- **Graphics Output**, consists of all NSL graphic libraries for instantiating display frames, canvases and graphs;
- **Graphics Input** consists of NSL window controllers to interact with the simulation and control its execution and parameters.

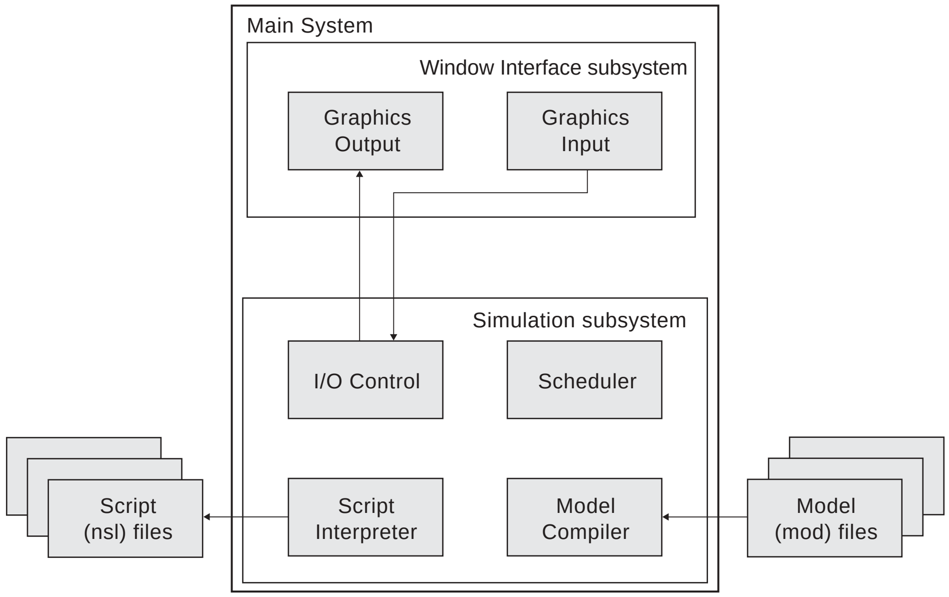

Figure 1.10
NSL Simulation System composed of the Simulation and Window Interface subsystems.

Schematic Capture System
NSL supports development of models by explicitly programming the code for each module as well as visual modeling by using the *Schematic Capture System* (*SCS*). The Schematic Capture System facilitates the creation of modular and hierarchical neural networks. SCS provides graphical tools to build hierarchical models following a top-down or bottom-up methodology. 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/Module Libraries, users will benefit by being able to create a rich variety of new models without having to write new code. When coming to view an existing model, the schematics make the relationship between modules much easier to visualize; besides simplifying the model creation process. To create a new model, the user places icons on the screen representing modules already available and connects them to provide a high level view of a model or module. As modules are sum-

moned to the screen and interconnected, the system automatically generates the corresponding NSL module code. The success of this will obviously depend on having good modules and documentation.

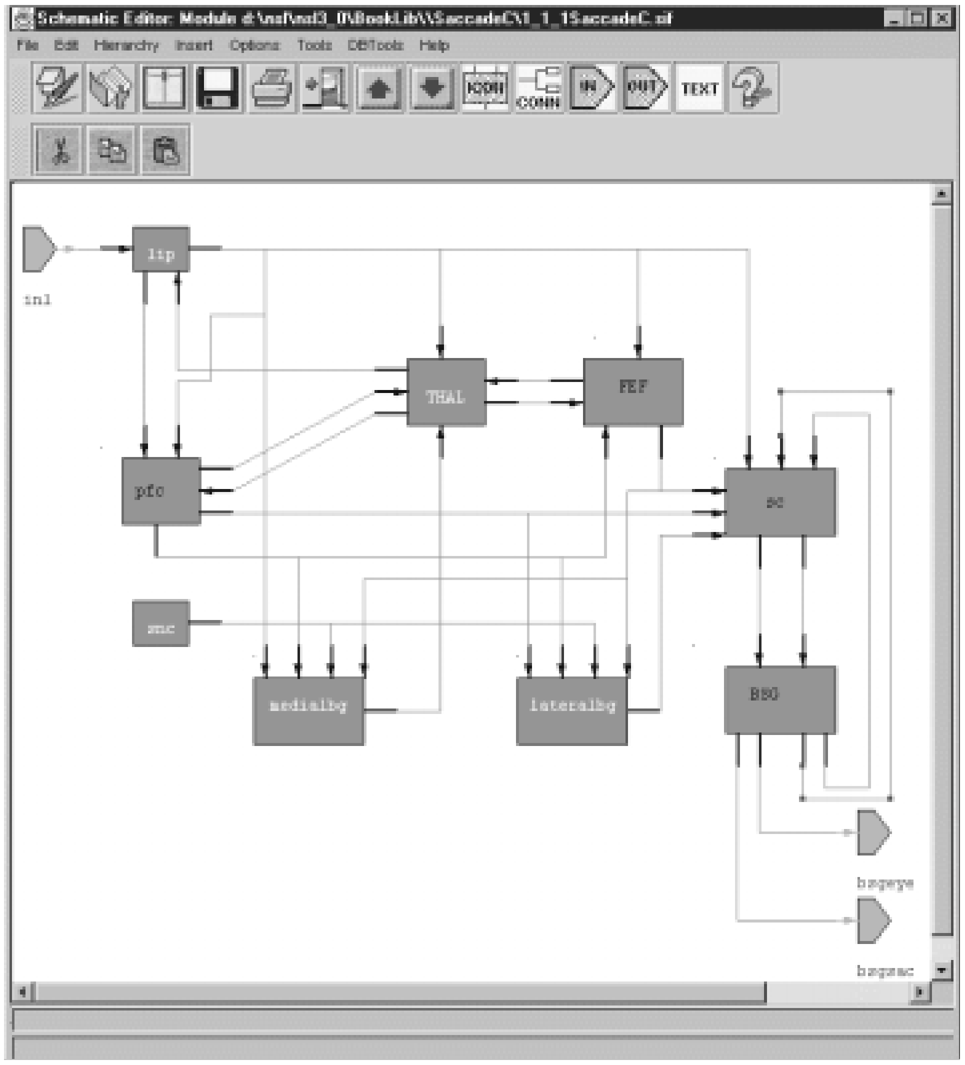
Schematic Editor: Module d:\nsl\nsl3_0\BookLib\\SaccadeC\1_1_1SaccadeC.sif
File Edit Hierarchy Insert Options Tools DBTools Help
ICON
CONN
IN
OUT
TEXT
in1
lip
THAL
FEF
pfc
sc
snc
medialbg
lateralbg
BSG
bsgeye
bsgsac

Figure 1.11
Schematic Editor showing the Crowley Top Level Saccade Module. Thin lines describe connections among sub-modules while thick lines describe entry (with arrows) and exit points to and from modules.

Figure 1.11 shows a schematic of the top level of a model. The complete schematic describes a single higher-level module, where rectangular boxes represent lower-level modules or sub-modules. These modules can be newly defined modules or already existing ones. Thin lines describe connections among sub-modules while thick lines describe entry (with arrows) and exit points to and from modules. Pentagon shaped boxes represent input ports (when lines come out from a vertex) and output ports (when lines point to a side) for the higher-level module whose schematics is being described.

SCS also provides many of the library functions that are necessary to organize and manage the modules including model and module version management as overviewed in the following section. More details of the Schematic Capture System are described in chapter 4.

Model/Module Libraries

Models and modules developed under NSL are hierarchically organized in libraries. NSL supports two library structures. The first is called the basic hierarchy while the second structure is known as the extended hierarchy built and maintained by the Schematic Capture System (SCS). Both are shown in table 1.2. The difference between the two is how modules are managed. The basic organization does not give a version number to modules only models. The extended one 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.

Library Organization						
Basic Hierarchy			Extended Hierarchy			
nsl3_0			nsl3_0			
Library Name			Library Name			
Model Name			Model or Module Name			
Version Number			Version Number			
io	src	doc	io	src	exe	doc

Table 1.1
NSL model library hierarchy organization for the Basic Hierarchy on the left and the Extended Hierarchy on the right.

There are several reasons for maintaining both systems. In the extended one, the user can experiment with different versions of a module shared among a number of models. Typically, larger models will share modules thus needing management by the SCS system. For the basic structure (not using SCS) it is easier to manage all of the module files in one directory, **src**. Additionally, 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.

Basic Hierarchy

In the general organization of the basic hierarchy levels in the tree correspond to directories. The root of the hierarchy trees 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 identified each by its corresponding numerical *version*; (version numbers start at 1_1_1). At the end of the directory hierarchy, the last level down contains the directories where the actual model or module files are stored: input/output files (**io**), source module files (**src**) and documentation (**doc**). The **io** directory stores input and output files usually in the form of NSLS script files. The **src** directory contains source code that needs to be compiled 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 this book where originally developed using the basic system. table 1.2 illustrates the directory hierarchy for the basic book models described in chapters 2 and 3 in the book. Note that we actually have two versions of the Hopfield model; one where we illustrate the use of scripts for input, and another for illustrating the use of input and output modules.

Basic BookLib											
MaxSelectorModel			HopfieldModel						BackPropModel		
1_1_1			1_1_1			1_2_1			1_1_1		
io	src	doc	io	src	doc	io	src	doc	io	src	doc

Table 1.2
The basic hierarchy organization for the book models.

Extended Hierarchy
In the extended hierarchy, the directory structure for the library is almost identical to the basic one except for the fact that each module is versioned, and there is an extra **exe** directory. There may be multiple libraries, and it is up to the model builder to decide what modules and models will go into each. Also, each library may contain multiple *models* and *modules*, identified by their corresponding name. Each model and module must have a unique name. Also, each model and module is then associated with different implementations identified by its corresponding numerical *version*, (version numbers start at 1_1_1). Obviously, many versions of a model or module may exist in a library, thus we identify versions using a version identification number composed of three digits denoting the model or module release number, revision number, and modification number, respectively. All numbers are initialized to 1. At the end of the directory hier-archy, the last level down contains the directories where the actual model or module files are stored: input/output files (**io**), source module files (**src**), documentation (**doc**), and the executable files (**exe**). Typically the **io** and **exe** directories are empty except for model directories. In table 1.3, we illustrate the MaxSelectorModel hierarchy previously shown in table 1.2 in the basic architecture and now shown with modules in the extended library.

Extended BookLib																							
MaxSelectorModel				MaxSelector				MaxSelectorStimuli				MaxSelectorOutput				ULayer				VLayer			
1_1_1				1_1_1				1_1_1				1_1_1				1_1_1				1_1_1			
io	src	doc	exe	io	src	doc	exe	io	src	doc	exe	io	src	doc	exe	io	src	doc	exe	io	src	doc	exe

Table 1.3
The extended library structure for the basic book library showing one of its models, the MaxSelector, and its children.

SCS manipulates the model and module library allowing 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” the most current version of the module would be used instead and whenever there is a more recent version of the module, that version will be used. This is called “using the **floating** version.” Each individual library file stores *metadata* describing the software used to create the corresponding model/module.

1.5 Summary
In this first chapter we have introduced modeling and simulation of neural networks in general and in relation to NSL. We also gave an overview of the NSL system components including a description of the technology used to build the system as well as simulate models using NSL.

Notes

1. Figure 1.9 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 assump-tions and simulation results can be explicitly compared with the empirical data gathered by neuroscientists.