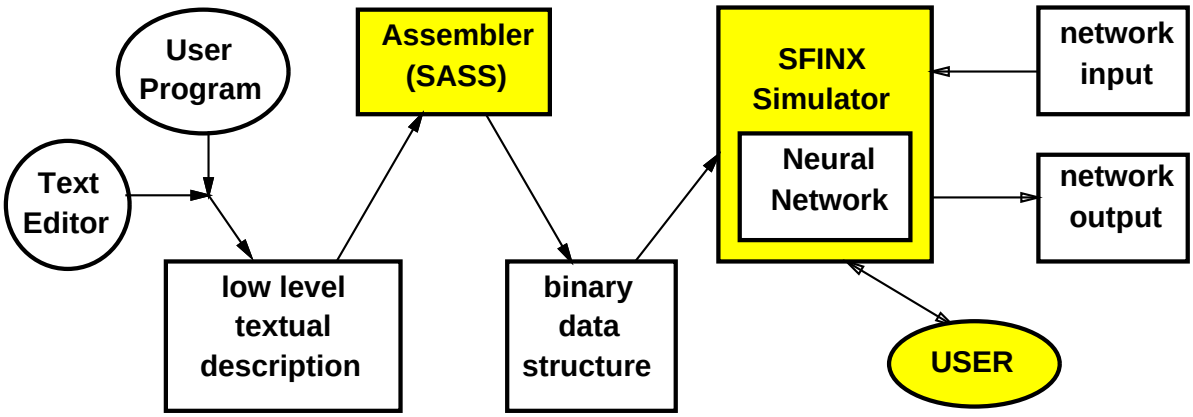
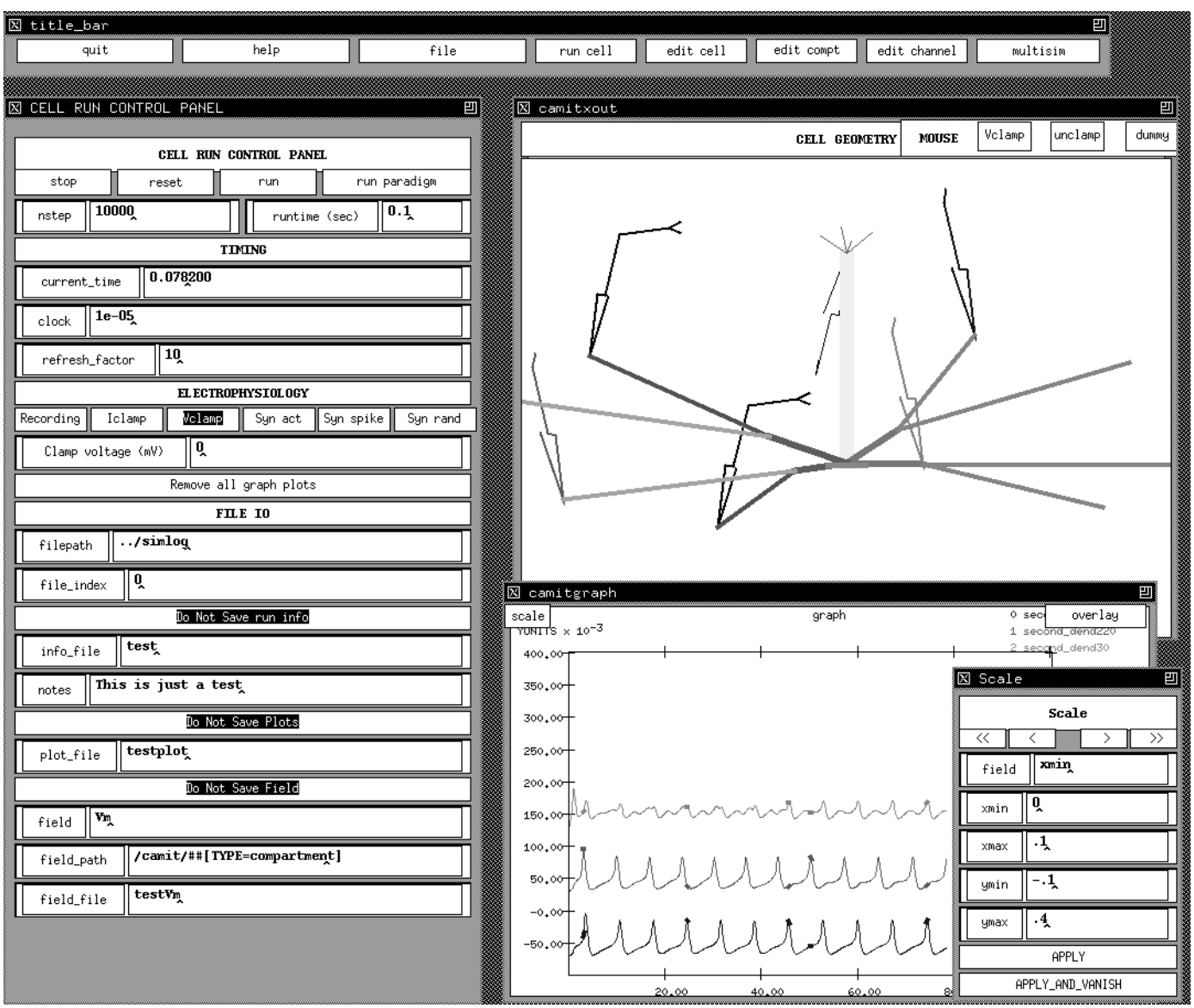
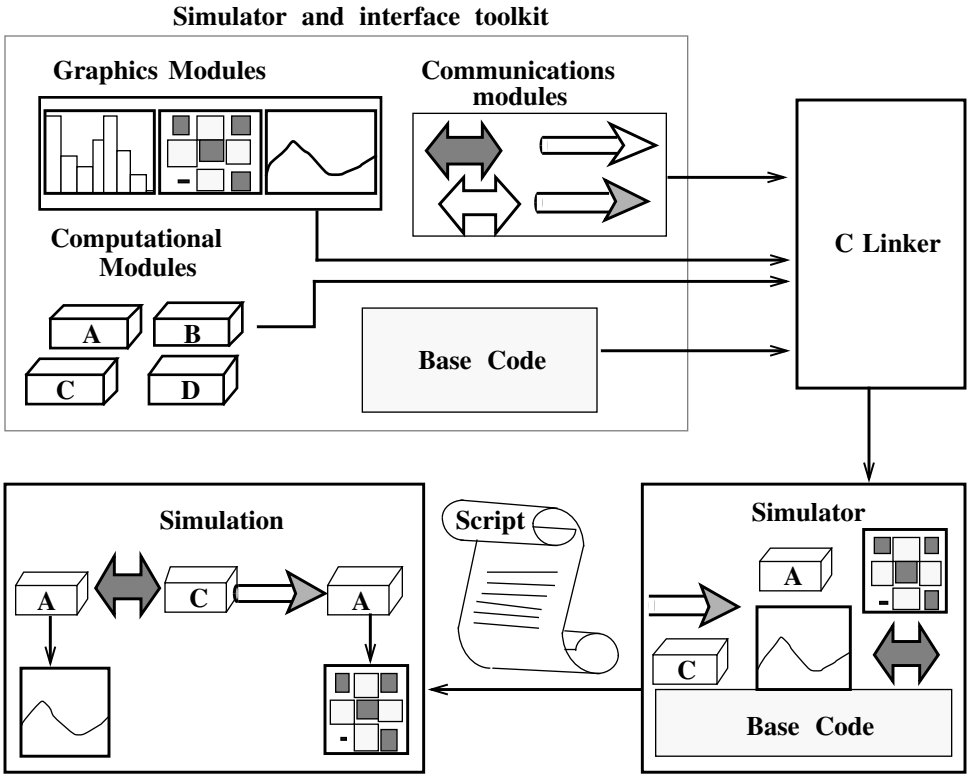
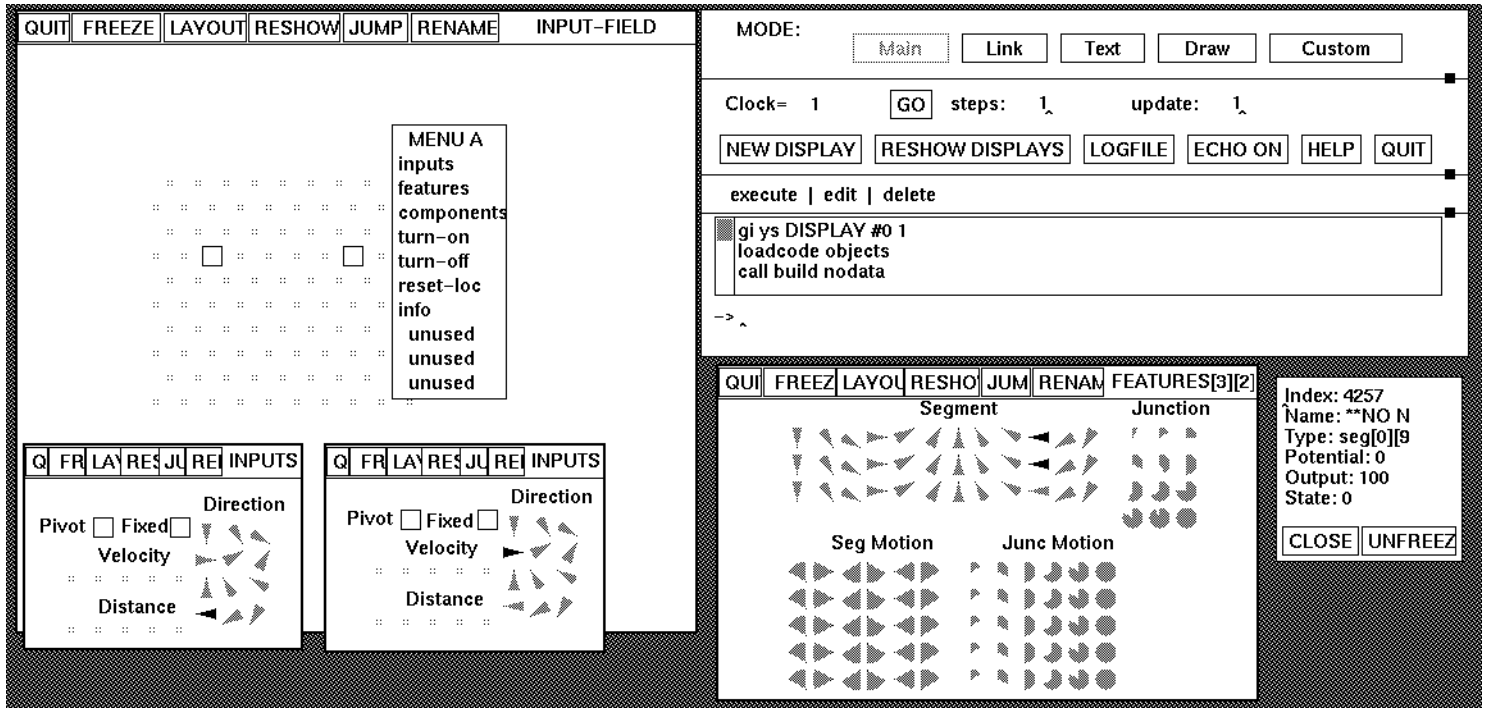
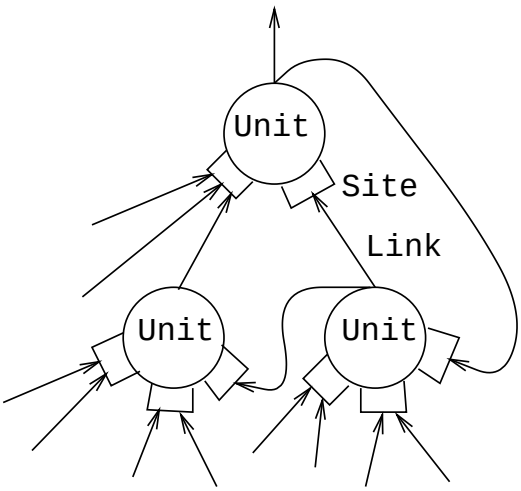
Command Purpose	RCS	GENESIS	SFINX
Add/Change/Delete elements and connections	Primary means of building up network is a conventional call to a 'build' function written in C. Interactive commands include: AllocateUnits <number> MakeUnit, AddSite, DeleteSites, MakeLink, DeleteLinks	create <element-type> <pathname> createmap (multiple copies) delete <element-path> sendmsg/deletemsg - send or delete messages between elements.	load filename (explicit networks must be 'assembled' into binary representation prior to use. No changes to the pre-defined network are allowed within simulations.
Set Node values	Out <UnitID><value> set output value also Pot, State, Weight, UFunc, SFunc, LFunc, Reset, delay	set <path> <field> <value> - set the field at path to value	en - for 'Explicit Network' will set or display one or more fields in specified unit(s) depending upon parameters.
Get or Display Network Values	List < link | set | unit UnitID > show values of the described item	show <path> <field> - list the value of a given element's field(s)	
Execute Simulation Cycles	go <stepcount>	Necessary to create a simulation schedule with AddSchedule, ClearSchedule and then use the command step <stepcount> to execute it.	runen <stepcount> for explicit networks and runfa <stepcount> for implicit networks.
Save/Restore the Simulation State	checkpoint/restore will save values but will not restore the network. save/load will retore values and it will rebuild the network itself.	No provision for saving or restoring is provided.	save/load restores the entire network, with the exception of function pointer values.
record and replay commands	log/read	logfile/include	no provision for recording exec <filename> reads commands
global user variables	Available in user-coded functions	int five = 5 float ten = 10.0 str hello = "Hi There!"	set - envirionment variables setu - user's variables usetu - delete user's variables
Name Conventions	Individual or subscripted names for Vectors and Arrays. Unit index can also be used. Ranges of units can be specified for unit index and subscripted names. In some cases, set names can be used. NameUnit command used to assign names.	Elements are referenced using their full 'pathname'. Wildcards may be used for portions of the pathname so that additional organization of the network may be established.	3 digit subscripts [z, y, x]. Subscript ranges may given as: num -- One specific value low: -- Greater than low :high -- Less than high low:high -- from low to high <nothing>-- All possible values
User Written Functions	call <funcname> <args> - Call user function using (argc, argv) for parameter passing.	addfunc <funcname> <sliname> - bind a user function to a script language command. Uses (argc, argv) also.	<funcname> invokes user functions User functions are included in simulator control lists, and use a command structure to parse the input and pass parameters.

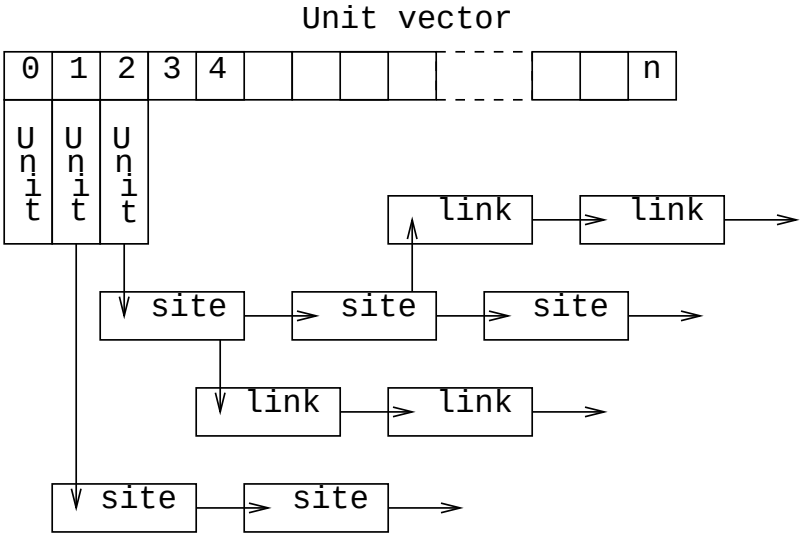

User Program
Assembler (SASS)
SFINX Simulator
Neural Network
network input
network output
Text Editor
low level textual description
binary data structure
USER


title_bar
quit
help
file
run cell
edit cell
edit compt
edit channel
multisim
CELL RUN CONTROL PANEL
CELL RUN CONTROL PANEL
stop
reset
run
run paradigm
nstep
10000
runtime (sec)
0.1
TIMING
current_time
0.078200
clock
1e-05
refresh_factor
10
ELECTROPHYSIOLOGY
Recording
Iclamp
Vclamp
Syn act
Syn spike
Syn rand
Clamp voltage (mV)
0
Remove all graph plots
FILE IO
filepath
../simlog
file_index
0
Do Not Save run info
info_file
test
notes
This is just a test
Do Not Save Plots
plot_file
testplot
Do Not Save Field
field
Vm
field_path
/camit/##[TYPE=compartment]
field_file
testVm
camitxout
CELL GEOMETRY
MOUSE
Vclamp
unclamp
dummy
camitgraph
scale
graph
overlay
YUNITS x 10-3
400.00
350.00
300.00
250.00
200.00
150.00
100.00
50.00
-0.00
-50.00
20.00
40.00
60.00
Scale
Scale
field
xmin
xmin
0
xmax
.1
ymin
-.1
ymax
.4
APPLY
APPLY_AND_VANISH


Simulator and interface toolkit
Graphics Modules
Communications modules
Computational Modules
A
B
C
D
Base Code
C Linker
Simulation
A
C
A
Script
Simulator
A
C
Base Code


QUIT
FREEZE
LAYOUT
RESHOW
JUMP
RENAME
INPUT-FIELD
MENU A
inputs
features
components
turn-on
turn-off
reset-loc
info
unused
unused
unused
INPUTS
Direction
Pivot
Fixed
Velocity
Distance
INPUTS
Direction
Pivot
Fixed
Velocity
Distance
MODE:
Main
Link
Text
Draw
Custom
Clock= 1
GO
steps: 1
update: 1
NEW DISPLAY
RESHOW DISPLAYS
LOGFILE
ECHO ON
HELP
QUIT
execute | edit | delete
gi ys DISPLAY #0 1
loadcode objects
call build nodata
->
FEATURES[3][2]
Segment
Junction
Seg Motion
Junc Motion
Index: 4257
Name: **NO N
Type: seg[0][9
Potential: 0
Output: 100
State: 0
CLOSE
UNFREEZ


Unit
Site
Link
Unit
Unit


Unit vector
0
1
2
3
4
n
Unit
Unit
Unit
link
link
site
site
site
link
link
site
site

Rochester, NY: University of Rochester, Computer Science Department

Joyce C.W. (1990). *Network Data Structures and Representation in the Simulation of Neural Networks* (Master's Thesis). Knoxville, TN: The University of Tennessee

Messrobian E., Stiber M., & Skrzypek J. (1989). *UCLA SFINX - Structure and Function in Neural Connections* (included with SFINX distribution). Los Angeles, CA: University of California at Los Angeles, Machine Perception Laboratory

Wilson M.A. (1989). *Documentation for GENESIS and XODUS* (Only available to registered GENESIS users). Pasadena, CA: California Institute of Technology

Wilson M.A., Bhalla U.S., Uhley J.D., & Bower J.M. (1989). GENESIS: A System for Simulating Neural Networks. (reprint included with GENESIS distribution) In. D. Touretzky (Ed.), *Advances in Neural Information Processing Systems I* (pp. 485-492) San Mateo, CA: Morgan Kaufmann

The RCS simulator may be the easiest to use to construct new paradigms, since the predesigned structures can accommodate a wide variety of paradigms. The SFINX simulator does allow one to construct arbitrarily complex nodes, but it does not easily allow one to construct site and link structures that can be accessed by the other simulator functions. GENESIS provides the most flexibility in this regard, but at its usual cost of increased complexity in the description process. The GENESIS simulator allows arbitrary propagation delays between elements, as would be required for biological models. RCS allows integer propagation delays that can be used to construct recurrent networks. SFINX does not explicitly allow for any propagation delays.

Lastly, the simulators can be compared according to the speed of their network setup and execution. In the combined task of constructing the appropriate Hamming network description, setting up the simulator, instantiating the network, and running the simulation through forty iterations, the RCS simulator was the fastest. The primary reason for this is that RCS builds its simulation directly from the output of the network generator, while both SFINX and GENESIS must first create an intermediate description file. The intermediate description approach has the advantage that if many runs are to be made with the same network, it can be much more efficient to load the network from the intermediate description than from the higher-level description. The GENESIS simulator takes far greater time to build the network description and construct the simulator than either of the two others; however, this added time pays off in the execution speed. Both RCS and SFINX require one fewer cycle than the total number of layers in order to propagate the data completely from the input to the output layer. Every node is visited (i.e., checked for null operation) each cycle, even though only a single layer is active at a time. The more complex representation formed by GENESIS visits the nodes in the correct order, according to the simulation schedule, so that only a single cycle through the nodes is required to propagate the data from the input to the output layer. This results in a much faster execution speed for layered types of networks. The GENESIS approach could be improved by allowing the user to examine and modify the simulation schedule at run time. The RCS simulator seems to have a somewhat faster execution speed than the SFINX simulator (this may be partly due to the fact that the RCS simulator tends to access variables and functions with direct pointers, while the SFINX simulator tends to use array indexing). It is only fair to point out that the use of an instrument such as the Hamming decoder network represents a limited artificial test and may not be indicative of how the simulators would perform on other tasks. As only a brief synopsis was presented above, Complete performance details for the Hamming test is available in Joyce (1990).

Conclusions

The SFINX simulator's unique quality is the distinction it makes in network types, and in its preparation of a binary simulation structure prior to run time. The implicit networks are highly efficient for image processing applications (indeed implicit networks can be viewed more as an image processing tool than a neural network tool), while the explicit networks allow the flexibility for more-general applications. Unfortunately, the manual does not fully address the possibility of combining these two networks types in a single simulation that may provide the best of both worlds. The simulator appears to be highly focused on image processing applications given its implicit network structure and narrowly defined graphical interface.

The RCS simulator's strength is the clarity of its design approach and its manual. Also, given its version number and clarity, it appears to be the most mature of the simulators. A C programmer would be very comfortable extending the simulator to implement new models. It is an excellent tool for exploring general connectionist models which involve large numbers of simple neurons. Its graphical interface is well suited for connectionist models and is extremely easy to use.

The GENESIS simulator is the most flexible and object oriented of the three. It appears to have been designed primarily for the complex and detailed modeling of small biological systems. In this regard, the Hamming network may be less fit as a test instrument for GENESIS, as compared to its use for the other simulators. The complex representation used for neurons may limit the number of neurons that it can handle adequately, as the description file for the 16,433 node Hamming network was almost 5 Mbytes in size. One advantage of this complex representation is an increase in the execution speed. GENESIS suffers from inadequate documentation, an "annoyance fee", and a large learning curve is needed for one to become proficient in its use. In short, GENESIS is not for the casual user, but it will allow a serious user to perform detailed simulations of realistic biological models at many levels of detail.

Mike Plonski is with the Computer Systems Research Department at The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009. Chuck Joyce performed portions of this review while at the University of Tennessee. He is currently an independent consultant residing at 508A Park Ridge Road, Apt. 2, Durham, NC 27713

References

Cohen, H. (1989). How Useful Are Current Neural Network Software Tools? *Neural Network Review*, 3, (2)

Goddard, N.H., Lynne, K.J., Mintz T., & Bukys L. (1989). *Rochester Connectionist Simulator* (Technical Report 233 rev., included with RCS distribution),

interface. This touches the issues of how the simulator is programmed for new paradigms and how one can extend the command language. Table 1 outlines the basic set of commands used to establish and operate a network simulation for each of the simulators. These same basic commands are used whether via a command line or through a graphical interface. All three simulators contain provisions for making use of previously prepared command scripts. Both RCS and GENESIS make provision for logging commands. This is useful in that these command files can be edited and re-used as command scripts. It is surprising, particularly in light of the amount of time involved in setting up a large simulation, that GENESIS contains no provision to save or restore a simulation's current state.

Table 1. Simulator Command Language Comparison

Of equal importance to both the operation and programming components of the user interface is the method used to refer to elements of the network simulation. The RCS simulator allows the use of simple sequential index numbers that are convenient for programming purposes, as well as provides for the use of names (with optional subscripting of one or two indices) and sets of units more suited for interactive use. The system of addressing elements in the GENESIS simulation is very natural for Unix users. It provides the same abilities provided by the various name and set functions in RCS, within a single overall conceptual framework. However, each time a command is invoked, a full pathname must be typed, parsed and interpreted (or the user can set the "current element" to the object of the command). The use of pathnames in a programming context is awkward at best, and some more direct means of reference might be useful for this purpose. The three-range subscript-type reference method used in SFINX is an interesting compromise. It is easier for a human to work with than one might expect, and is not difficult to program. In addition, it can be consistently applied to SFINX's explicit and implicit networks, reducing the burden of learning diverse systems for these models.

RCS allows the user to create and manipulate the network directly by using the C language to call elements of the subroutine library. It also allows easy access to this subroutine library from the command language interface, by passing all subroutine arguments as command line parameters using standard (argc, argv) C language constructs. The SFINX approach varies in that it is somewhat more circumspect. The explicit networks are created separately, by the assembler, into an efficient binary representation of the data structure used to construct the network. Implicit networks can be constructed entirely by means of the command language, but the models that can be implemented in this fashion are quite limited. SFINX utilizes C-macro expressions to isolate the user somewhat from the underlying differences between the implicit and explicit network models. There are two ways of programming the GENESIS simulator. First, the script language interpreter (SLI) provides a much more powerful set of control constructs than are available in the command scripts of the other simulators. In addition, new classes and methods can be prepared by means of the object-oriented paradigm and the C language. However, this flexibility is not achieved without some cost. The interpretation of the script language takes up an amount of time that is not expended by the other simulators. The objective of the SLI may be to provide a higher-level language for individuals who have limited programming experience, but it is inadequately documented and not substantially easier to use than a standard language such as C. The object-oriented style is also hampered by inadequate documentation (this may change, as the manual is still considered preliminary), but is simple and direct once grasped. The creation of new classes, including such objects as the stopwatch used to time the simulation, are handled consistently.

All of the simulators have some X Windows graphics support to one degree or another. SFINX is the most limited as it allows one only to display the outputs of a layer as an image. RCS has direct support for Hinton-type diagrams, built-in drawing tools for annotating the display, and the capability to customize the mouse easily. It is also possible to link various existing X Window widgets into the RCS interface, and an example of how to do this is provided. The RCS graphics utilities are extremely easy to use and well documented. The GENESIS/XODUS graphics support should be viewed as an environment for constructing interfaces, rather than as a specific interface (as is the case with the other simulators). This has the strength that each network model can have an entirely different interface, as is vividly demonstrated by the sample simulations. Unfortunately, a sample graphical interface suitable for connectionist models (similiar to that of RCS) is not included, as the GENESIS is heavily slanted towards the detailed modeling of a small number of neurons. The flexibility of the GENESIS interface does not come without a price. It is highly unlikely that a casual user could construct a custom interface without a great deal of effort, simply because of the complexity involved and the lack of adequate documentation.

Unfortunately, none of the simulators come with a large or even a small library of predesigned network paradigms. All of the simulators have the necessary extendabilty to construct many of the existing common paradigms by using a combination of custom code and existing modules.

simulator is invoked. A command interpreter script file is used to load the image into the input buffer, open a display window, display the input buffer, apply the convolution node function to all nodes in the output buffer, and display the output buffer. Note that there are two ways to apply implicit functions to a buffer: the first applies the same function for every (y,x) node in the buffer, while the second uses a (y,x) function buffer to apply a different function to every (y,x) node.

As with explicit network functions, user-defined implicit network functions can be precompiled and linked into the run-time simulation environment. Note that the implicit node function must be capable of determining the pattern of connectivity (a receptive field) as positions relative to the current node, as there is no explicit storage for links in an implicit network. No text description file is required for implicit networks, but the user must keep track of what is loaded into each buffer and set certain variables to indicate input and output buffers for the implicit node functions. This can all be done quite easily at run time by using the command interpreter or script files. It appears that both types of network may be active in a given simulation; however, the manual does not address this possibility.

The command interpreter is used to control interactively the simulation and access network variables. One can add user-defined commands to the interpreter by specifying the appropriate "yacc"[4] code and recompiling the simulator. A particular node in SFINX (in either type of network) can be referenced by the command interpreter via the node identifier. This node identifier is a set of three integers, or ranges having the form [z_range, y_range, x_range], where a single value indicates a specific coordinate, and a range is expressed as "min:max." If a range is left empty,it signifies all possible values in that range.

During simulation, explicitly defined nodes are processed in the order in which they were defined, and the output for all nodes is updated only at the end of a complete cycle through the network. Implicit nodes are processed only when an implicit function is applied to the buffer that contains them. The sample backpropagation code (implemented as an explicit network) suffers from the same shortcoming as the RCS implementation; the user must specifically disable the node functions for all but a single layer for each network cycle. The number of cycles required to implement a forward and backward propagation is equal to two times the number of weight levels. During each of these cycles, the function pointers for each node must be changed to null or active, and every node must be visited to determine if its function is currently active. Note that implicit networks could be used to avoid much of this overhead for backpropagation; however, the user would be restricted to a fixed receptive field (such as for full connectivity) and each layer would have to be the same size.

The command interpreter allows one to open a display window (under X Windows) and display the output values for a layer (buffer) as a gray-scale image in that window. The pixel replication factor and the position of the image in the display window can be controlled by simulator variables. This display window can then be dumped to a printer. There are no other graphics capabilities, as the remaining graphics commands are all used to perform window housekeeping. Additional interpreter commands allow one to load/save buffer values, set/run explicit and implicit functions, examine/set variables, load/save networks, copy values between buffers, and run command scripts. On-line help for all commands is provided. It is clear from this interface and the implicit network description that the author's prime application area is image processing.

The implementation of the Hamming test network as an explicit network was straightforward. A simple modification of the network-generation program wrote a description of the test network to a text file, wrote a Unix script file to assemble the network, and finally prepared the SFINX command interpreter script used to run the simulation. As before, the entire series of tests was run by means of a Unix shell script that required no operator intervention. Also, the preparation of a user-defined explicit node-activation function for the Hamming network was very simple. No node-initialization function was required. Activation functions in explicit networks receive standard parameters consisting of a pointer to the current node, the previous output value and an array of current input values. The link weights are obtained from the node pointer by means of a predefined C macro. The activation function returns the node's output value. The explicit activation function created for the Hamming network is shown below as pseudocode:

```
enfn_threshold(node_pointer,old_output,inputs){
  sum = 0
  for all links coming into this node
    sum = sum + link  weight * link input
  if (sum > threshold) then return 1
  else return 0
}
```

Simulator Comparisons

The first criterion by which the simulators can be compared is the user interface. Two components of the user interface stand out in our examination of these simulators. The first is the simulator's basic operation: how good is the command set, how difficult is it to control the simulation. The second component is the programming

[4] Yet Another Compiling Compiler (YACC) is a standard Unix utility that is commonly used for command interpreters.

must be activated in that order. The functions to load the input and unload the output of the network and to operate the simulation itself were prepared in the script language. The style of this language is similar to C. These scripts invoked the C functions that provided and checked the input and output to the network.

For testing purposes, the network generator was invoked to produce the network description file. The simulator was then invoked with the description file, which included two other script files to define the network objects and establish the simulator functions discussed above. As before, the process was described in a Unix shell script, so that the entire testing sequence required no operator intervention.

SFINX

The SFINX simulator (Structure and Function In Neural conneXtions), was developed at the Machine Perception Laboratory of the University of California at Los Angeles (UCLA). The version of the simulator used for review (2.0) was released in November of 1989, with several minor patches applied since then. SFINX was an outgrowth of an earlier neural-network simulator effort at UCLA called PUNNS. In order to ftp this simulator, a license agreement must be submitted. The license agreement form and additional information can be obtained by sending electronic mail to "sfinx@retina.cs.ucla.edu". Upon receipt of the signed agreement, an ftp account and instructions on how to obtain the software will be sent.

This simulator currently requires color to operate the graphics interface, but it may be run without graphics; a new version (2.1) providing support for monochrome and 24-bit graphics is expected to be released shortly. Support for Sun, Ardent Titan, HP 300, and IBM PC RT machines is specifically mentioned; however, it should run on other Unix platforms as well. Specific graphics support is provided for Matrox VIP 1024, Imagraph AGC-1010P, HP Starbase, and X Windows. A complete manual is included with the distribution; however, it was very difficult to get the figures for this manual to print.

The original PUNNS simulator was apparently used with gray-scale images as input and specified its network structure by means of a "connectivity language." This connectivity language allowed the user to specify a node's name, activation functions, and connections to other nodes. The PUNNS simulator demonstrated that while explicit specification of each node yielded tremendous flexibility, the resulting network required large description files and increased load and initialization times. In designing SFINX, the authors decided to use a two-pronged approach to handle this tradeoff between flexibility and efficiency. They drew a distinction between two network types, choosing to call those networks that require node-by-node specification "explicit", while networks consisting of "two-dimensional layers of functionally identical nodes whose connectivity patterns are highly regular and fixed" are called "implicit" networks.

To create an explicit network, a simple ascii text description of each node and its patterns of connectivity is written by the user in a file. In some of the sample simulations provided with the distibution, as well as in the Hamming test network, a C-program is used to generate this text description. One of the goals listed by the authors is eventually to provide a high-level description language that can be used to automatically generate these text description files, however there is no specific mention of when such a language might be available. The text file is then passed through a SFINX assembler program (SASS) to produce a binary representation of the network in a file which can be loaded directly into the simulator at runtime. This binary file is basically a large data-storage area that contains space for all of the network values, along with some initial values. This reduces the runtime overhead of parsing and interpreting a descriptive language, allows for a single large allocation of memory, and somewhat mitigates the problem of the size of the specification files. Figure 5 shows a description of the process for creating an explicit network simulation.

Figure 5. SFINX Overview

Each node in an explicit network has (1) a data structure which contains a node id (an integer tuple [z,y,x] to conform to implicit network-naming conventions), (2) indices to an initialization and activation function, (3) one output register and a variable number of general registers, and (4) links to other nodes. A small number of sample initialization and activation functions are provided and user-defined ones can be easily written, as was the case for the Hamming network. User-defined functions are compiled separately and linked into the simulator prior to run time. At run time, the command interpreter is used to specify which functions are to be used for which nodes. The simulator then copies the indices to the appropriate functions into the node data structures.

In an implicit network, each node is not associated with a separate data structure, rather, the network is implicitly represented by a three-dimensional [z,y,x] buffer array structure. This structure can best be viewed as a stack of two-dimensional (y,x) arrays, where the buffer number (z) refers to a specific array. Each buffer can either hold data registers or function pointers. The simplest node representation would be a single register to represent the node activation level. One of the sample simulations illustrates the use of implicit networks to perform the convolution of an image with a small kernel (i.e., a small receptive field with fixed weights). Space for two single-register buffers (input and output) are allocated when the

When an object is instantiated, the specific types of actions and messages the object responds to must be defined. The object methods for the SEGMENT and PROJECTION classes in the Hamming network are shown below as pseudocode:

```
HardThreshold (Object_Pointer, Action) {
  If (Action is INIT or RESET)
    then object activation and output = 0
  Else If (Action is PROCESS) then
    If(Activation > threshold) object output = 1
    Else object output = 0
}

ProjectOutput (Object_Pointer, Action) {
  If (Action is INIT or RESET)
    then object state = 0
  Else If (Action is PROCESS) then
    Obtain new state from message
    If (new state = 1) then for every connection
      Add connection weight to target activation
}
```

The GENESIS simulation is built up in a tree structure of elements. These elements, consistent with the object-oriented approach, may be elements of the network (such as neurons, synapses, and so forth) or they may be objects that perform the functions of the graphic display, or provide disk I/O and other system services, or they may be "neutral" elements used merely to organize the tree. This tree structure mimics the Unix file system format for addressing elements of this tree. For example, the base, or root of the tree is "/", and an element further away in the structure of the tree might be addressed as, "/branch1/branch4/element3". References can use wildcards, and GENESIS understands the concept of "current position" within the simulation tree structure. A simulation is organized into a hierarchical structure of elements, and it must be constructed in a manner that creates that structure from the top down. To facilitate the process of setting up a simulation, once an element (possibly involving several objects, messages and connections) has been created, other elements or arrays of elements may be instantiated by means of the element name. The new elements will contain the same substructure of objects, messages, and (internal) connections.

The control of the simulation cycle is vested in a simulation schedule that describes, in order, the network elements and the actions they are to perform. In this way, simulation cycles may update their output immediately. A secondary aspect of this method of organization is the concept of a task. Multiple schedules may be created, each run separately or simultaneously. Potential uses of this feature might include the creation of a "learning cycle" to alternate with the feed-forward cycle for back-propagation networks, or the precise simulation of input stimulus in a biological model.

GENESIS provides for "clocks" to be associated with each element of a network for the precise control of propagation delays and time-based simulations. The simulator may be controlled through the graphical interface or from a command line, but the preferred method appears to be through command scripts that make use of its own script language interpreter (SLI) which provides a full set of operators, basic program constructs such as "if" clauses, and function-definition constructs.

The XODUS interface to the graphics routines is at a much lower level than that of the RCS simulator. While this allows one to build extremely flexible interfaces, it may require a great deal of code to do so. There is no set graphical interface per se, as each simulation model is responsible for designing its own interface by using elements from the graphics modules. Figure 4 is a screen dump from one of the tutorial simulations provided to construct and study the actions of a single biological neuron. The mouse is used to place recording electrodes as well as voltage and current clamps into various compartments of the neuron, as shown in the upper right-hand part of the figure. The control panel is used to set various parameters and run the simulation. Plots of the membrane potential at the recording electrodes are shown in another window. Portions of the cell body change color to indicate the membrane potential. This gray-scale rendition does not do full justice to the visual effect and meaning that can be provided by a full-color display.

Figure 4. GENESIS Graphical Interface for Neuronal Simulation Environment

In order to implement the Hamming decoder test program as a GENESIS simulation, a number of things had to be done in addition to constructing the classes already discussed. As the simulator operates primarily on script files, the network description had to be written as a script file that could be read by the simulator. To construct the network, a single element was instantiated and then copied into position (multiple copies can be created by a single command). The connections for each element and the weight for each connection were established, each requiring separate commands. Finally, a simulation schedule was prepared.

To time the simulation process, a "stopwatch" object was created and was given the actions START and STOP during the actual simulation cycle. Between these actions, the other active elements were invoked with the action PROCESS. Only input units were not active, but each element in the network is composed of a SEGMENT class object and a PROJECTION class object, which

released in June 1990. The current version is the result of a four-year programming effort and has been used for laboratory courses at Caltech and at the Marine Biological Laboratory in Woods Hole, MA. To obtain the distribution via ftp, it is necessary first to obtain an ftp account by using the "telnet" utility to login as user "genesis" on "genesis.cns.caltech.edu" (131.215.135.64). Follow the instructions to obtain the ftp account, which can then be used to transfer the software. Mail can be sent to "genesis@caltech.edu" for additional information on acquiring the simulator.

The simulator has been tested on both Sun workstations and DECstations, and the graphical interface can run under X Windows or DECwindows. Unfortunately, the distribution release does NOT contain a user's manual. The only overview of the simulator is provided by copies of three papers included with the distribution. As indicated in the distribution *readme* file, "the free version of GENESIS ... is intended to give interested parties a taste of the GENESIS simulation package". In order to obtain the manual, one must become a registered user by paying a one-time "annoyance fee of $200" to join the GENESIS Users Group (BABEL). It also appears that updated versions of the simulator will be available only to those users who have been "annoyed". Registered users also have access to additional simulator components, bug reports, and a user's bulletin board.

While the tutorial simulations are somewhat self-explanatory and can be run without the manual, GENESIS is so complex that one would have little hope of constructing a new simulation without the manual. One of the reviewers joined the user's group and obtained a manual in order to aid in constructing the simulation of the Hamming network. The distribution *readme* file indicates that the manual is in an "early stage", which is an accurate description of its current state. This current state is particularly disappointing in light of the fact that the manual was originally written in May 1989 and that one must join the user's group in order to obtain a copy. The primary shortcoming of the manual is that it does not adequately address all of the complexities that make up the GENESIS simulator.

One of the guiding principles behind the design was a desire to facilitate the simulation of interacting processes on many different scales. Other design objectives include modularity and flexibility, expandability, the use of a "high-level" network-specification language "accessible to people with limited programming experience," user interaction through a flexible "friendly" graphical user interface, and an implementation as efficient as these constraints will allow. Concerns for device independence and the ease with which the simulator can be moved to parallel computing architectures are also mentioned as factors. The GENESIS simulator is definitely slanted towards the detailed biological modeling of a small number of neurons. One of the sample simulations uses several thousand lines of script files to provide a general-purpose simulation tool for single neurons. GENESIS can be used to simulate connectionist models, which involve large numbers of simple neurons, but this is not its forte.

The GENESIS simulator is object-oriented and written in the C programming language. Object-oriented programming in C is not, perhaps, as simple as it might be in a language such as smalltalk or C++, which have features designed specifically to support the paradigm, but it is possible.[3] GENESIS provides several basic classes of modules, and the simulation is built up from objects instantiated from these classes. Three basic units are used in network simulations: computational, communications, and graphics modules. The computational modules are combined into "elements," which comprise the basic unit of a simulation. These elements are then connected by communications modules. The graphics modules, based on X-window widgets, can be used to construct the user interface. Figure 3 indicates an overview of the stages used to construct and run a simulation.

Figure 3. GENESIS Overview (from Wilson, et al. 1989)

For example, the computational modules constructed for the Hamming test network are made up of two specific classes, which are used together to represent the "neuron." The SEGMENT class acts like the neuron body and is analogous to the "unit" in RCS. The PROJECTION class passes the activation of the segment forward to the SEGMENT class objects which receive it by using communication modules of the CONNECTION class. The PROJECTION class differs from the RCS "site" in that it passes activations forward, where RCS fetches activations from connected units. A second type of communications module provides "messages" among classes of objects. A message in this example is used to pass the output of a SEGMENT to its PROJECTION within each element. Messages differ from instances of the CONNECTION class in that they are not subject to propagation delays. The functions, or methods, that drive each object must accept one of a specific group of actions (typically, "INIT" to initialize, "RESET" to clear previous activity, and "PROCESS" to perform their primary function). Each object's methods must also account for any specific type of message that the object may receive or send.

[3]Further discussion of object oriented programming is beyond the scope of this paper. For a concise description of object oriented programming in C, which strongly resembles the style used in GENESIS, see: Brumbaugh, D. (1990). Object-Oriented Programming in C. *The C User's Journal*, 8, 113-122.

callable from the command interface. Once the basic data structures are understood, it is extremely easy for the user to construct custom functions and subroutines. The simulator allows one to dynamically link in new subroutines at run time from the command interpreter. One of the sample simulations gives an example of dynamically linking a routine that attaches an X Window scrollbar widget to a variable. The variable (e.g. learning rate) can then be controlled by using the mouse and scrollbar.

Reference to units in RCS takes several forms. Each unit is assigned an index number (its subscript in the unit vector), which can be used to refer to it in user programs and on the simulator command line. Units and sites can also be assigned ascii names, which can be used to refer to them. Site names are used in commands when units may have more than one site. A field of units can also be designated as a vector or array, and then an individual unit can be referenced by use of the field name with appropriate subscripting. Units may also be designated as members of one or more sets; these sets may be operated on by using the common set operations of union, intersection, and so forth.

The graphical interface has a rich set of tools that can be used even if one has no knowledge of X Windows programming. There are several types of windows (control, display, information, and layout), which are used to display the simulation results as shown in Figure 2. There are also facilities for developing custom pop-up menus for the mouse. The graphics interface is designed primarily for controlling and displaying a simulation. One could probably construct a means to build a network graphically by using the custom pop-up menus and mouse; however, this would be extremely inefficient and difficult as the interface was not designed for this. The control window (the upper right-hand corner of Figure 2) allows one to access the command interpreter and maintains a scrollable window of previous commands.

Figure 2. RCS Graphical Interface (modified from unpublished figure included with distribution)

A novel approach is used to display graphically values within the unit structures or the link weights. A family of icons is used to display all values that fall within some specified data range. For example, if a five-member family is used to represent values from 0 to 1, then the second member of the icon family would be displayed if the value fell between 0.2 and 0.4 . The upper left-hand display window in Figure 2 shows the use of Hinton diagram icons to display information, while the other display windows show user defined icons. Only Hinton-type icon families and a gray-level icon family is supplied with the simulator; however, standard X Window utilities can be used to define custom icons. Note that the simulator cannot display all the link weights at one time, but only the weights to or from a specific node. The graphics interface only functions in monochrome, so that it is not possible to display a field as an image (except by using gray-level or dithered icons).

It was necessary to construct unit and site functions for the Hamming test simulation. The link function is set to NULL and a flag can be set to avoid the overhead associated with a NULL function call. Unit functions are passed the unit function as an input, and the individual sites can be accessed by descending a linked list. A similar procedure is used to access the links entering a site. The RCS unit function, which performs a hard threshold, and the site function, which performs a weighted sum, are shown below as pseudocode:

```
UFthreshold (unit_pointer) {
  if (sum of all_site values for unit > threshold)
  then unit output = 1
  else unit output = 0
}

SFweightsum (unit_pointer, site_pointer) {
  sum = 0
  for every link into this site
    sum = sum + link weight * linked_node output
  site value = sum
}
```

A simulation cycle basically involves looping through the unit vector, performing each unit's link, site, and unit functions in that order. The output vector is not updated until the a complete simulator cycle is concluded. For asynchronous operation, a specified percentage (100% = synchronous updating) of the units is randomly selected for updating during an individual cycle.

The Hamming decoder network could be automatically constructed under the RCS simulator, either by generating a C language *build* subroutine or by writing a command language script. The former method is the one advocated in the manual and was the approach taken. In addition to the functions shown above, a user command interface routine was also written; this operated the test simulation by calling for specific input, loading and unloading the network, and operating the timer functions. The entire test sequence was run from a series of Unix shell scripts, without operator intervention.

GENESIS

The GENESIS (GEneral Network SImulation System) simulator and its companion graphical interface XODUS (X-based Output and Display Utility for Simulators) was written by the Division of Computation and Neural Systems of the California Institute of Technology (Caltech). The version of the simulator used for this review (1.1) was

RCS

The RCS simulator (Rochester Connectionist Simulator) was developed by the Computer Science Department of the University of Rochester. The version of the simulator used for review (4.2) was released in October of 1989, with several minor patches applied since then. It was originally implemented in LISP and was gradually translated to the C programming language for performance reasons. RCS uses the license terms of the Free Software Foundation.

The simulator is available for anonymous ftp from the "pub/simulator" directory on "cs.rochester.edu" (192.5.53.209). Additional information is available by sending electronic mail to "simulator-request@cs.rochester.edu". The installation instructions were quite simple and compilation proceeded without problems, except for the minor annoyance caused by the installation routine assuming that some of the directories were pre-existing. Both RCS and GENESIS are easier to install if one has "superuser" privileges that allow the user to install the simulator components in the "preferred" system directories.

Specific installation instructions for Sun, MIPS, and DEC workstations are provided. The simulator has also been ported to many other Unix platforms, and an unsupported port to the Macintosh is included in the distribution. The current version supports SunView and X11 graphical interfaces. The X11 interface was new with Version 4.2 and is somewhat more powerful than the SunView version. Electronic mailing lists exist for users and bug reports. The simulator distribution includes an outstanding manual.

Connectionist models are constructed from three basic building blocks: units (nodes), sites, and links. Each unit can have multiple input sites and each site can have multiple links from other units. Units are stored as a structure array, while the sites and links are stored as a linked list of structures. Figure 1 illustrates both the model and data-storage representation. Each structure contains several fields (13 for units, 7 for sites, and 6 for links). In addition to these structures, there is a separate output vector, which is used to buffer the output field for the units; one of the fields in the link structure is a pointer to the corresponding element in the output vector. This buffered output allows the simulator to be run with either synchronous or asynchronous node updating. A special version of the simulator can be compiled to provide additional output buffering, which can be used to simulate propagation delays.

Figure 1. Network Structure for RCS (modified from Fig. 2.1 Goddard, et. al 1989)

While this data structure representation is extremely memory expensive, it provides enormous flexibility. The authors clearly indicate in the documentation and by the design of the simulator that flexibility was their primary objective. Each field in the structures (with the exception of a user-defined data field) is predefined, and contain such elements as a numeric value (e.g. output, weight), a pointer to a function, a pointer to a nametable entry, a pointer to other structures, initial/reset values, set indicators (groups of units can defined to belong to a set), and various flag bits. Each of the unit, site, and link structures contains its own function pointer and user-defined data field.

If all the link inputs are to be processed identically, it is generally more computationally efficient to use null link functions and allow the site function to perform all of the processing. The link functions are necessary only if one needs different types of links connected to a single site. For example, one link could compute a weighted input, while another link could compute a distance function. The user-defined data field is 4 bytes, which can be used to store such things as a pointer to a user-defined structure. This allows one to construct arbitrarily complex representations, although the predefined structure fields are more than adequate for most models. It is hard to image a connectionist model that could not be implemented by means of the structure functions and user-defined data field.

The command interface allows the user to set various system/data parameters; cycle the simulator; log commands to a file; run command scripts; access various graphics utilities; and create/delete units, sites, and links. Unfortunately, there does not appear to be any command to save/load only the unit activations to a file without saving/loading the entire data structure. All structures are dynamically allocated, so the net size can vary throughout the simulation. Basically, the simulator functions as a subroutine library. The command interface allows the user to access the subroutine library by passing the arguments using the standard (argc, argv) C format. The subroutines can also be accessed from user-written code. The entire simulator may also be called as a subroutine from other programs written in C, LISP, or MIT's Scheme.

An entire network can be constructed and run using the command language; however, the preferred method is for the user to write a C subroutine (nominally called *build*), that will construct the network, and then to use the command interface to call this routine, passing any appropriate arguments. This approach to designing the command interface is extremely flexible, since it allows direct access to user-defined code while the simulation is being run. Unit, site and link functions are called by the simulator with appropriate structure pointers when the network is cycled; consequently, these functions are not

ules are pre-designed and the user simply has to specify how to connect them together. While a full-custom design will generally run faster than one constructed by means of a simulation environment, the slight increase in speed usually does not justify the added effort required. These environments also allow a user to design new modules easily; these modules can then be incorporated with the existing modules to construct highly specialized simulations.

The three simulators discussed in this document are the RCS, GENESIS, and SFINX simulators. The examinations are based on the versions indicated, which may not be the latest versions available at the time of this publication. Each was acquired via "ftp"[2] and installed on a Sun 3/60 workstation at the Computer Science Graduate Laboratory of the University of Tennessee's Computer Science Department in Knoxville, and on a Sun 4/65 at The Aerospace Corporation. For those without ftp access, all three simulators are available on magnetic media for a nominal charge (typically $50 - $150) from the addresses listed at the beginning of this article. Note that all three simulators will run on a variety of other Unix platforms besides the two Sun workstations listed above.

Each of the simulator environments reviewed here provide all the features discussed above. Full source code (in C language) is provided for each simulator, which allows a brave and ambitious researcher the option of modifying the basic nature of the simulation environment. This option is not available for the commercial equivalents of these simulators (e.g. the AXON Neural Description Language by HNC). Several sample simulations of a tutorial nature are provided with each of the simulators. However, none provide a full or even partial library of the popular network paradigms that one might find in the commercial simulators. Each of the simulators is installed in a similar fashion. One or more configuration files are edited, according to enclosed instructions, to conform to the local computing environment, and the Unix "make" utility is used to compile and install the simulators. Each simulator allows the user to specify, at compilation time, the variable types (e.g. integer, float) used in the simulation. An X Windows graphical interface is provided with each simulator. All of the simulators can be run without graphics, as was the case when the relative speed of each simulator was compared. Also, in some cases alternative graphical interfaces are available; however, only the X Windows interface was considered for this review. Finally, on-line help is provided with each of the simulators.

[2]The term "ftp" stands for file transfer protocol, which is a means of obtaining and transferring files between computers linked by an international computer network collectively know as the Internet. To obtain more information concerning ftp or the Internet, please consult operators of your local computing facility. The format used for site addresses and electronic mail addresses used in this report conforms to that currently used on the Internet.

In order to provide a consistent basis for comparison, we decided to implement a Hamming single-bit-error correction network (Joyce 1990) on each of the three simulators. The Hamming code is an error-correcting code that provides for recovery from single-bit errors that may occur during transmission of data. The digital logic that implements this error correction is well understood, and a custom C program was written to construct a linked list of units and connections, which simulated the operation of the digital logic. It should be noted that there is no learning in this network as the units basically function as digital logic gates and the connections determine the inputs to the gates. The Hamming network is constructed to detect and correct single-bit errors, or to indicate the existence of uncorrectable errors. The activation function employed in the Hamming network performs hard thresholding against a sum of weighted inputs. To illustrate the differences between the three simulators, we custom-coded this activation function into modules suitable for each of the simulators. A Hamming network was designed for each of fourteen different input code lengths, ranging from 4 to 2048 bits; this required developing fourteen corresponding networks whose sizes ranged from 45 to 16433 nodes. For each network a simulation was run; this included the program startup, network instantiation, and forty processing cycles from input to output. In order to measure the computational efficiency of each of the simulators, timing routines were added.

An individual discussion of each simulator constitutes the main body of this review. Information on how to obtain the simulator, a brief history of the simulator, an overview of the simulation environment and features, as well as a brief discussion of the implementation of the Hamming network is presented. This material is drawn from both the documentation supplied with the simulators and the personal experiences of the reviewers (in using these products). Also, specific phrases taken from the simulator documentation are generally quoted throughout these sections. Finally, comparison and conclusion sections are provided at the end of this review, in an attempt to address the strengths and weakness of the individual simulators.

It should be noted that one of the reviewers had prior experience in using the RCS simulator before performing this review; a fact that may have biased these observations slightly. Neither of the reviewers have any background in biological modeling which may have caused a slight bias against the GENESIS simulator. Unfortunately, space limitations do not allow us to list the many individual contributors who developed these fine simulators and so generously made them available to the neural-net research community. On behalf of the neural net community, we are pleased to thank them collectively for a job well done.

RCS, GENESIS, and SFINX: Three "Public-Domain" Simulators for Neural Networks

RCS (4.2). c/o Peg Meeker, Computer Science Dept., University of Rochester, Rochester, NY 14627

GENESIS (1.1). c/o Prof. Jim Bower, Computational Neural Systems Program, Caltech 216-76, Pasadena, CA. 91125

SFINX (2.0). c/o Prof. Josef Skrzypek, Machine Perception Laboratory, 3532 Boelter Hall, UCLA Computer Science Dept., Los Angeles, California 90024

Reviewed by Mike Plonski[1] & Chuck Joyce

Introduction

The last decade has seen a huge resurgence of interest in the study of artificial neural networks by a wide variety of disciplines. There have recently been offered many commercial products that purport to allow the student or amateur to "explore" the world of neural networks. A recent review by Cohen (1989) summarizes three of the more popular commercial simulators. The purpose of this review is to provide some insight into three of the more ambitious common "public-domain" simulation environments: how they are acquired, how the networks are described and controlled, and how the simulators might best be used. A significant portion of this review has been condensed from a more complete review contained in a Master's Thesis by C. Joyce (1990), with additional commentary and analysis provided by M. Plonski.

So called "public-domain" software has been around long enough to invoke several negative impressions. First, one has the image that a product that is freely distributed cannot be worth much, a premise shattered in the field of computer software by the Free Software Foundation. Second, one has the suspicion that freely distributed software is poorly documented and unsupported. There is, of course, some basis for this perception, as the author who releases software at no cost has little incentive to maintain that software. Finally, there is the perception that freely distributed software is not subject to license or restrictions. In many cases this is true; however, many sources of freely distributed software do require licenses and place restrictions on their work. For this reason, the term "publicly available" will be used throughout this report, rather than the more common term "public domain," which implies the lack of any such restrictions.

Research in neural networks has always been closely tied to simulations of the action of a network. Many researchers have developed neural simulation software and have made that software publicly available at no cost, so that others could duplicate and advance their work. This software can be classified into three basic groups: (1) a "demonstration" program, which is intended to demonstrate a single type of network model and limited aspects of that network's behavior; (2) a "special-purpose" simulator, which is usually intended to allow a broader range of experimentation for a particular set of network models; and (3) a "simulation environment," which supports a wide class of network models as well as the ability to construct new models. These environments typically allow users to create their own network paradigms, activation functions, and learning methods. The three simulators examined in this review are all of the "environment" category. Appendix A of Joyce (1990) contains a fairly complete list (approximately 20) of various publicly available neural simulation software, as well as information on how one can obtain copies for individual use.

A simulation environment has many advantages over constructing, *from scratch*, a custom software simulation for a particular neural network model. The simulation environments typically allow the user to describe the network model in a higher level of abstraction and in a more compact form than if one were to construct the entire model from a standard programming language. The environments usually have built-in graphics capabilities and an interactive command language. The simulation environments also enable one to set up *control scripts*; these allow the user to make multiple simulations of a basic model, with minor variations in parameters. Also, many researchers may not have the time or the programming skill required to construct a complete network simulation from scratch.

In many cases, designing a network within a simulation environment is analogous to designing a semicustom VLSI chip, where all of the computing elements or mod-

[1] This work was supported, in part, by Aerospace Sponsored Research