ICSIM: An Object-Oriented Connectionist Simulator

Heinz W. Schmidt*, Ben Gomes
ICSI, Berkeley, California

Abstract

ICSIM is a connectionist net simulator being developed at ICSI and written in Sather. It is object-oriented to meet the requirements for flexibility and reuse of homogeneous and structured connectionist nets and to allow the user to encapsulate efficient customized implementations perhaps running on dedicated hardware. Nets are composed by combining off-the-shelf library classes and if necessary by specializing some of their behaviour. General user interface classes allow a uniform or customized graphic presentation of the nets being modeled.

The report gives an overview of the simulator. Its main concepts, the class structure of its library and some of the design decisions are sketched and a number of example nets are used to illustrate how net structure, interconnection and behavior are defined.

1 Introduction

In a highly exploratory field of research like that of artificial neural nets, simulation seems to be the only prototyping technique combining sufficient flexibility with acceptable cost. Flexibility is essential, due to the different mathematical models underlying neural nets, the different network architectures and applications and also due to the experimental character of most research projects (cf. e.g. [12, 15, 9, 1, 16]). Different simulators serve different purposes ranging from modeling bio-chemical processes in the human brain (e.g. [17]) to developing structured connectionist models of artificial memory, recognition and reasoning processes (e.g. [8, 11, 3]). Efficiency is equally important; the simulation of the massively parallel nets, for instance in real-time speech recognition, may take hours on sequential machines.

Existing simulators like the Rochester simulator [8] or Genesis [17] lack the ability to deal with nets in a modular fashion supporting the partial reuse of existing prototype nets. Moreover they often started off as simulators with a textual interface and a graphic interface for visualization of net behavior and performance is either missing or put on top of the textual dialogue interface. This compromises extensibility. For instance, if new types of nets are added, the command language and the related modules may also need to be extended.

Some form of interactive, incremental prototyping is necessary to allow the user to change the representation and/or structure of nets during a simulation. This is particularly

*on leave from: Inst. f. Systemtechnik, GMD, Germany

important in case of long simulation runs. Otherwise, in a non-incremental environment, these long runs tend to repeat essentially the same problems as the edit-compile-debug cycle in the batch-orieted style of programming in the seventieth. For instance, in the development of the Genesis simulator this need was specifically addressed by an intermediate shell language interpreter [18] and in the Rochester simulator a special linker/loader was developed to support a kind of dynamic binding of binary code for the same reason.

We believe that most of the above requirements related to extensibility, reuse and incrementality can be met by an object-oriented design of the simulator. In the decentralized view of objects, functionality is organized along the dimension of data types. The uniformity of the 'message passing metaphor' is independent of whether messages are implemented by procedure calls or real message passing in the sense of communications between processes. Message passing lends itself not only to the support of command-language-like interfaces in an incremental prototyping environment but also to the integration of separate tools. Finally the abstract data type paradigm embodied in object-oriented languages provides acceptable high-level mechanisms that have advantages over special purpose languages for declaring net topology and interconnections (e.g. [10, 3]).

In the near future, we expect network sizes in the range of some hundreds of thousands of units. Although larger nets are conceivable theoretically and, as nature suggests, realistic, the technology to deal with heterogeneous nets of such size does not yet exist. For nets of some hundred thousands units, efficiency must be addressed by parallel hardware combined with a dedicated selection of data representations to reduce storage requirements and the resulting paging and garbage collection overhead. Data structure selection and hiding of the chosen data representation is also naturally addressed by the abstract data type approach inherent in object-oriented languages. Parallelism of heterogeneous collections of objects comes naturally in object-oriented terms and while the hope is that an object-oriented approach to parallel simulation will lend itself to the development of massively parallel applications, this is the topic of ongoing research in the field of concurrent object-oriented languages.

Summarizing, the design of the ICSIM simulator tries to achieve the following goals:

1. Support a library of standard building blocks including novel structures like structured and heterogeneous networks, shared weights, different kinds of parallel paradigms etc.

2. Provide flexibility and extensibility in network architecture and behavior (including learning).

3. Create simulations that are modular and as easy to understand (declarative) as possible.

4. Permit quick incremental changes to the networks in a protyping environment.

To achieve these goals we have chosen to

1. shift the conceptual focus from units to nets, and from global and sequential execution to local and asynchronous execution;

2. combine flexibility of object-oriented design with efficiency by virtualization of structures (e.g. virtual connections), parallel execution, dedicated processing, e.g. downloading nets to dedicated hardware[1].

The current focus of the design is on the structure of the class libraries and, to some extent, on the separation and interaction between the simulator proper and the user interface objects. The simulator is written in Sather[13], after initial prototype implementations in Common Lisp and Eiffel.

The design tries to assist the specialist scientific community in teaching and research in artificial neural networks. We envisage three types of 'users' to ICSIM that possess increasing programming skills:

1. A graphic *point-and-click interface* is to allow the ICSIM beginner and maybe the non-programmer expert to get acquainted with the available objects, tools and their functionality. This interface is fairly limited. It supports only a fixed set of objects and some standard ways of combining them.

2. A *simple library* of standard classes with a meaningful set of independent classes and a coherent default functionality allows combination of off-the-shelf classes and simple specializations in terms of 'single-point' redefinition. If this is done well, most users should be able to do their simulations at this level.

3. skilled object-oriented programmers can use extension and inheritance to implement radically different algorithms based on an *advanced library* with a set of dedicated classes including new ways of visual presentation.

Due to the importance of extensibility and reuse and the limitation of 'point-and-click' interfaces, our initial design focuses on the users in the second and third class.

2 Overview

Two worlds can be distinguished in ICSIM: *models* and *views*. Models are the primary object of interest in a simulation. Different kinds of networks, units and special approximation and learning methods fall in this category. Views are the first-level objects that a user sees and manipulates to deal with models: textual and graphical presentations, operations on them that change the state of models, their behavior or their presentation.

Views have the character of objects to the extent that they are the medium through which models are seen. At the same time they have the character of tools to the extent that models can be manipulated through them.

In ICSIM we have chosen to separate models and views down to the level of instances for three reasons:

1. Massively parallel objects justify a many-to-many relation between models and views. In order to reduce complexity, separate views may filter different aspects of a single chosen model.

[1] such as the Ring Array Processor (RAP) developed at ICSI

2. The amount of presentation related (graphical) information must be reduced to a minimum in the presence of many thousands of instances; and presentation information and techniques deserve an encapsulation of their own to keep orthogonal issues separate in the design;

3. We expect parallel execution of models in which server processors have local memory to hold model instances combined with a user interface running on a workstation that holds view instances.

2.1 A simple example: Computing the XOR of two random inputs

Fig. 1 shows a small net of four units computing the XOR of two random inputs.

The hidden unit in the center of the net is a boolean threshold of 2 and the output (top) a boolean threshold (of 0.5 by default). The example demonstrates connection functionality on the lowest level (wiring units one by one) and the choice of a specific computation mode.

Skipped psfile=ps/xor.ps

Figure 1: Xor net

This model is implemented by two small classes representing the net (Xor_Net) and a view of the net (Xor_View).

Upon creation, the interactive text view class creates, initializes and then runs the interactive view. All the simulator behavior, i.e. command prompting and interpretation, net access and so forth are inherited from the superclass Text_View_Net and its ancestors. The main purposes of the class Xor_View are to select the proper type of view and to specialize the type of net to be simulated (here Xor_View).

class Xor_View **is** Text_View_Net{Xor_Net}; **end;**

An (Xor_Net) is a Net composed of units (cf. below). Any_Unit is the most general unit class. This means, in the context of this class, we can call only the most general subset of operations on units but we are free to create arbitrary units as components of the XOR_NET – descendents of ANY_UNIT.

As part of the creation, this class builds the corresponding units by means of inherited initialization routines and then wires them. For simplicity, we use integers here to 'name' the units. Units 0 and 1 represent the input units, unit 2 the hidden and unit 3 the output unit. The inherited *create* routine calls back the routine *create_Component* redefined to 'declare' the proper type of units and finally calls *finish* which completes the creation. Instances of class BOOL_RANDOM_UNIT are created as input units; a boolean unit with a variable threshold of type VAR_BOOL_UNIT is chosen as hidden unit and the threshold is defined as 2.0; and finally a boolean unit with constant default threshold of 0.5 is made the output unit. A *parameter map* (PMAP) object details the creation specifics, here the selected weights depicted in Fig. 1 (1.0 and −2.0). Parameter maps allow to associate typed values to symbols (or keywords). In the example below, for instance, the REAL value *-2.* is associated to the symbol *initial_weight*. Particularly during initialization ICSIM uses PMAP as a uniform and dynamically extensible way of parametrization to provide objects at different levels of a structured net with access to common structural information.

```
class XOR_NET is NET{ANY_UNIT};

 finish is
   c1: PMAP::create.r(initial_weight,1); c2: PMAP::create.r(initial_weight,-2.);
   unit(2).connect(unit(0),c1); unit(2).connect(unit(1),c1);
   unit(3).connect(unit(0),c1); unit(3).connect(unit(1),c1);
   unit(3).connect(unit(2),c2);
 end;

 private create_Component (i: INT): ANY_UNIT is
   switch i
   when 0..1 then res := BOOL_RANDOM_UNIT::create;
   when 2 then res := VAR_BOOL_UNIT::create(2,2.0);
   when 3 then res := BOOL_UNIT::create(3);
   end;
 end;

 step is serial_Step end;

end;
```

This defines a running model. All the necessary functionality including means to describe the current state of the net, to manually inspect and update single units are inherited. The corresponding functionality is retrieved, compiled and linked when the root class of this simulation, class XOR_VIEW, is compiled.

2.2 Models

Models are built from units and nets. We speak of units (rather than neurons) to stress the artificial character of these objects and their difference from physiological models of neurons. In ICSIM, units and nets are characterized by their *structure* and their *behavior*. The structural aspects include *composition*, *interconnection* and *state*. Behavior is subdivided into *computation* and *learning*. Together these aspects make up the *functionality* of the respective objects.

2.2.1 Structure

Nets are either composed of units or, recursively, of other nets. Informally units can be thought of as atomic nets. The common functionality of nets and units is captured in the class Any_Model. This class defines the protocol for interconnecting and triggering the computation steps of nets and units. Any is the most general of all the simulator related classes. It encapsulates various general routines, like access to some global information, printing routines, persistency and the creation protocol.

Typically the state and behavior of a net is understood as the combined distributed state and behavior of its component subnets or units. In this sense, a net is ultimately defined in terms of its lowest-level units and most of the functionality of units carries over to nets in a natural way. For instance we may say that a net performs one computation step when all of its units performs a single computation step. Most of the ICSIM classes are compositional in this sense. The hierarchical structure of nets plus the interconnection structure between its components uniquely extends the low-level unit behavior to the high-level net behavior.

The complete story is more complex, when we introduce intermediate levels of subnets and consider asynchronous computation modes. For instance, the implementation of a net's behavior is not independent of the data representation of its state. A specific choice of a compact data representation on the subnet level will typically require a specific way to implement the computation of the net. Also, modeling a specific computation mode of a net may require departure from a pure compositional semantics. For instance, structured connectionist nets [7, 14, 11], that model semantic networks, will usually require special computation modes to implement mechanisms like variable-binding, token passing or inference steps. Our hierarchy of net classes is designed to allow the user to form subclasses in such a situation. There are a number of intermediate classes in the hierarchy whose purpose is the definition of a corresponding abstract data type only and which do not yet freeze a specific data representation for their subclasses.

2.2.2 State of computation

The activity of units is represented by their activation levels. Units have an internal activation level which we call *potential* and an externally visible activation level called *output* (visible to other units). The potential is usually some real-valued function of the net input from other units. For instance, for many of our instantiable library units the potential is the weighted sum of inputs. Different types of units may have different additional attributes to model more complex states and state transitions. In some models [6, 14], a unit has a *mode* represented by a value in a small range of integers that represent different phases of computation. We subsume all the related accessor functionality of classes under the general term *state*.

The output of a unit is some function of its potential such as the result of some squashing or threshold function, cf. Fig. 2.

Skipped psfile=ps/synthetic.ps

Figure 2: Unit synthesis

For the class ANY_UNIT, this value is computed but not stored. In other classes (TWO_PHASE_UNIT and its descendents), *output* is a stored attribute separate from *potential*. This separation is used in a sequential simulation of simultaneous steps. Units can compute their potential without affecting the 'simultaneous' computations in other units which 'see' the unchanged output of the previous state. Only when all units in the simultaneous step have computed their potential, will all of them change their output (cf. Section 2.2.4 below).

2.2.3 Interconnection

Units receive input signals from other units via directed connections. These connections may carry weights representing the connection strength. The interconnection functionality of units includes primitives to connect them and to initialize weights. For learning, specific unit classes embody training rules which define how weights are changed during the computation cycles of a net in response to signals from a net's environment. The interconnection of a net thus represents the knowledge or state of training of the net. Because the weights of a connection directly influence the activation of its target unit only, one can view the weights of the input connections of a unit as part of the overall unit function.

Nets can be interconnected in many different ways. A single net can exhibit many different interconnection structures. Therefore different connection procedures are, in general, supported by the same type of net. In contrast to this, units, viewed as atomic nets, have a single *connect* procedure.

There are different ways to distinguish different interconnection structures. In the most general case, the internal interconnection structure of a net, once it is built, is a spaghetti bowl of indistinguishable connections.

One simple means of imposing structure on interconnection is *partitioning* the net on the instance level. A specific component of a net may be connected only to some other specific component, this 'knowledge' about interconnection is built into the connection routines and selection routines of the net.

Another, structural mechanism is a *type-specific partitioning* of the net. In such a partitioning only certain types of components are connected with each other. In this case the respective types can encapsulate the knowledge about their interconnection. The net connection and selection routines can use a uniform protocol for inquiring components about their connections.

Finally, *sites* can be used for partitioning the connection structure. Sites correspond to different types of connections. Semantically, they may be viewed as a partitioning of the net adjacency matrix. In the distributed representation that we have chosen for many of the simulator classes, units hold a corresponding vector of the connection matrix. A site then groups the corresponding part of such a vector.

In ICSIM sites also encapsulate the representation of weights. Moreover, different sites may have distinct functions for contributing to the computation of a unit. Sites thus lead to a *structural* and *functional* decomposition of units. However, in ICSIM, they do not involve a *temporal decomposition.* This means, unit steps are the atomic steps in our discrete simulation.

Units and sites have a single output while the output of nets is that of perhaps many units. Units and sites therefore obey a common information transmission protocol captured by the class ANY_SINGLE_OUTPUT (cf. Fig. 3). Based on this protocol it is possible to compose sites and units in a uniform way to form a *site tree* where the output of one site is fed into another site and so on until the whole input is integrated and worked out by the receiving unit. The lower part of Fig. 2 alludes to this possibility. The resulting structure is loosely analogous to dendritic trees. The output of a single unit can also be fed into such a site tree many times via multiple connections[2]. Although semantically more complex, from the viewpoint of modeling power and also of implementation, this scheme has advantages. It is more powerful than simple unidirectional connections between pairs of units because it allows one to express complex non-linear dependencies by means of simple standard site and unit functions in terms of structural arrangements.

In the simplest case, however, a unit has a single site and the library includes site classes which enforce simple connections between units.

The interconnection structure of a net is an important aspect of the way it computes and generalizes its trained behavior. There are so many variants of interconnection types and structures that they deserve a high-level means to describe them. The *connection specifications* that parametrize the various connection procedures are part of the initialization specifications (*init_pmap*). For instance *x_connect* (short for 'cross connection') connects two nets by connecting all units of the first to all of the second net (cf. also Fig. **??**). A connection specification can 'tell' *x_connect* to skip certain connections, to connect with some probability only and so on. It also 'tells' the initial weight to use and for multi-site units it tells which site to connect to.

In this way users can customize interconnection structure on a high level – a more

[2]Physiologically, many neurons have multiple connections with adjacent neurons and the activation of these synaptic connections depends on learning such that there is the possibility of many different non-linear interactions between the same two neurons[2].

Skipped psfile= ps/output-object.ps

Figure 3: Any_Single_Output: Common functionality of units and sites

convenient way than programming in terms of the internal connection primitives of the different net classes.

To interpret a connection request at the various levels in the hierarchy, ICSIM supports general routines ANY_MODEL::*connectp*, INPUTS::*which_site*, and ANY_LINK::*initial_weight*.

At the net level the boolean function *connectp* will be called to determine whether two objects are to be connected. The result may depend on a probabilistic variable to allow for a randomly dense interconnection structure or it may depend on the type of subnets and units to be connected. Then, at the unit level, the site – if any – will be determined and finally, at the site level, the initial weight – if any. In this way a connection specification describes which connections to 'wire', to which sites to connect and what initial weights to choose.

Different connection specifications have different regimes to choose sites and initial weights. For class MULTI_SITE, for instance, we assume that a net sequentially builds its different types of connection structure site by site. A connection specification used in this process can be updated by the net (*set_current_site*) whenever one type of connection (site) is built.

The following example shows hierarchical nets and the use of different connection primitives. Our problem consists of coloring the map shown in Fig. 5, such that neighboring regions do not receive the same colors (white, green, red and blue). We have chosen the four color problem as a 'typical' local constraint problem.

More formally, we wish to find a total function *color* from regions to colors with the property that for two regions x and y in the *neighbor* relation *color(x)* differs from *color(y)*.

Our coding in terms of interconnection structure is obvious when we consider the following reformulated specification: We wish to find a *binary relation color* between regions

Skipped psfile=ps/con-spec.ps

Figure 4: Connection specifications customize interconnection structure

and colors, that is *total*, *unique* and satisfies the neighbor constraint above.

We now represent each region by four units, one for each color. If a unit's activation level is high, the corresponding pair (region,color) is considered to be a member of the color relation. The constraints are represented by inhibitory connections, shown in Fig. 6, as follows: The colors of one region are mutually exclusive (*complete_connect*) and the same colors of neighboring regions mutually exclude each other (*bus_connect*).

Choosing different inhibition strengths for the two types of exclusions and a particular nondeterministic unit function we let the net randomly walk through its combinatorial state space. The net has many fixpoints, each representing a solution.

We choose a weak inhibition between the same colors of neighboring regions (-1) and a strong inhibition (-10) among the colors of the same region, at least n times the weak inhibition, were n is the maximal number of direct neighbors in the net.

The four-color net (COL4_NET) is hierarchically built from regions (COL4_REG). A COL4_NET creates its regions as subnets and 'declares' the neighbor relations (part of the *create* routine) which are 'compiled' into the corresponding interconnection structure (*bus_connect*).

Each region is a net composed of COL4_UNITS. It creates its four color units as part of its own creation and interconnects them internally.

```
class COL4_NET{REGION} is NET1D{REGION};

 constant default_init_pmap: PMAP := PMAP::i(size,10);    -- ten regions

 neighbor(i,j: INT) is -- connect two regions as neighbors
   subnet(i).bus_connect (subnet(j),weak_inhibit);
   subnet(j).bus_connect (subnet(i),weak_inhibit);
 end;

 finish is    -- declare neighbor relation
```

Skipped psfile=ps/col4graph.ps

Skipped psfile=ps/col4map.ps

Figure 5: Region map and neighbor graph

```
    neighbor(0,1); neighbor(0,5);
    neighbor(1,2); neighbor(1,6); neighbor(1,5);
    ...
  end;

  private create_component(r: INT): COL4_REG is
    res := REGION::create_default; res.set_name("Region ".copy.i(r));
  end;

end;


class COL4_REG{U} is NET1D{U};

  constant default_init_pmap: PMAP := PMAP::i(size,4);

  finish(ignore: PMAP) is complete_connect (strong_inhibit); end;

end;
```

The connections specifications *weak_inhibit* and *strong_inhibit* are constant connection specifications like in the previous (Xor) example.

2.2.4 Behavior

The computation of a net consists of many update *steps* of its units. A single unit step has two phases, internally. The first phase *computes* the input received by the unit. This

Skipped psfile=ps/col4net.ps

Figure 6: Interconnection structure

input determines the internal *potential*. The second phase internally *posts* the potential so that it becomes visible as the unit *output* to connected units.

We refer to this combination of *compute* and *post* as a discrete unit *step*. Most often *compute* and *post* will be deterministic functions, often monotonic and usually differentiable. Sometimes it is useful to consider non-deterministic unit functions, i.e. units whose outputs depend on some random choice or models a specific probability distribution, i.e. a stochastic function. For instance, the compute procedure of *random units* sets the potential to some random value. Accordingly we call a unit *deterministic* if its function is deterministic and otherwise *nondeterministic*.

We distinguish *serial* and *parallel* computation.

Serial computation can be *deterministic* or *random*.

In *deterministic serial* computation the structure of a net defines a 'natural' order in which components step. For instance, a serial computation of a net may consists of a series of *step*s of its unit in the order in which they are structurally arranged in the net. Similarly, in this computation mode, in the step of a layered net, one subnet completes its step before the next one starts its step.

In the *random serial* computation, we randomly choose the component that is to step next. The different components are assumed to have a uniformly equal chance to precede any other component. The law of large numbers suggests that with increasing number of steps each component eventually gets a chance to step. This computation model assumes that the steps of two different components can always be temporally separated, i.e. temporal measurements can be arbitrarily fine and interferences of two components in between successive measurements can be neglected. These simplifying assumptions are often adequate.

For some models, tight temporal relations are essential, including real-time dependencies with bounded delays. There are three types of parallel computation modes: *synchronous*, *fair asynchronous* and *unconstrained asynchronous*. All parallel computation modes involve simulation of *real* parallelism. This means, units may update *simultaneously* without interfering spontaneously (limited signal speed). Intuitively, this means that the output of some unit cannot affect that of a simultaneously updating one.

The synchronous parallel computation mode (of a subnet) represents the tightest form of temporal coupling among the parallel modes: all units step simultaneously in parallel. This means all units compute their potential and then all units post some function of the potential to their output. Thus the output of a given unit's step can only affect the computations of other units in a subsequent step.

In the asynchronous computation modes some units may 'step ahead' of other units. This is similar to serial random computation. However, an arbitrary number of units updates simultaneously. Consider two units u and v where v receives some input from u. If u and v update simultaneously, v is not directly influenced by u's step. However, if u happens to step ahead of v, v will 'see' the current output of u. If u steps ahead multiple times before v updates, intermediate outputs of u may be lost. In the fair asynchronous mode, no unit can step ahead of any other unit more than a fixed number k of steps. Every unit has a fair chance to update eventually. In fact, its update step can be predicted in a fixed temporal interval determined by k.

All of these parallel modes are represented by special cases of what we call *bounded asynchronous* computation. We associate a *synchronization bound* k with a net, defining the maximal number of times a single unit can step ahead of other units. A synchronization distance 0 obviously corresponds to the synchronous mode. A positive integer value k guarantees fairness, i.e. each unit will eventually step but up to $k-1$ signals may be lost between two connected units. And the value ∞ corresponds to the unconstrained asynchronous model.

Many connectionist models rely on convergence of the net, i.e. the net function (defined in terms of its units' functions) is supposed to reach a fixpoint in which the output $\mathbf{x} = \mathbf{f}(\mathbf{x})$, where $\mathbf{f}$ is the net function composed of the unit functions f_i and $\mathbf{x}$ is the net state vector composed of the unit state values x_i. If $\mathbf{f}$ is the single-step function, i.e. determines the next state $\mathbf{f}(\mathbf{x})$, and the above equation holds in a given state $\mathbf{x}$, the net output has settled (is therefore statically stable). For recurrent nets it may be useful to consider $\mathbf{f}$ in the above equation as the function defined by a finite sequence of net steps. Even if the single-step function does not have a fixpoint, a multi-step function $\mathbf{f}$ may have a fixpoint. In this case the net oscillates (and is therefore dynamically stable). Net and unit classes have the specializable predicates *deterministicp* and *fixpointp* to help the simulator search and determine such fixpoint states. For a unit the predicate *fixpointp* represents the local fixpoint condition $x_i = f_i(\mathbf{x})$, in other words, the state (x_i) of unit i will not change (does not require updates) if the state of other units (in x) does not change. This covers the cases where units have ranges of input in which their function f_i behaves nondeterministically and also units the state of which may settle only after a number of updates due to internal state.

Specializations of all this standard functionality are possible and available. For instance the most general class of units, ANY_UNIT, does not distinguish between the two phases of a step while TWO_PHASE_UNIT does. The user may design subclasses of ANY_UNIT instead of subclasses of TWO_PHASE_UNIT, for instance, when it is clear that a simulation of 'real' parallelism is not important but an arbitrary serialization will do. Also when it is clear that the subclasses will always be executed on physically parallel processors there is no difference between these two classes.

For learning, units can be clamped, receive error signals, accumulate them and *self_adapt* their connection weights to accommodate for the error values accumulated so far. As part of this adaptation, they *feedback error* signals to their own inputs. In this way errors are recursively fed back towards the input units. Currently the back-propagation rule is the only learning rule implemented, cf. the corresponding routines of the class BP_UNIT.

The corresponding basic procedures of units are again driven by the net; the driving routine of the net is also called *self_adapt*. For instance in a *layered back-propagation net*, the corresponding net receives some error signals at the output layer (*teach_output*) and then *self_adapts* by letting the units adapt to the error signals according to the back-propagation algorithm starting from the output layer and working back to the input layer successively.

We conclude this section by completing the four-color example.

The output of a color unit (COL4_UNIT) is non-deterministic. When it receives weak inhibition only, a unit can still be activated with a certain probability. This is expressed in the feature *compute*. *unif_rnd* is a random number generator that returns a uniform random number between 0 and 1.

```
class COL4_UNIT is SIGMOID_UNIT;

  compute is
    potential:=20*(accumulated_input - RANDOM::uniform s̈trong_inhibition)+.5;
  end;

  fixpointp: BOOLEAN is
    in: REAL := accumulated_input;
    res := ( in <= strong_inhibition and output < 0.3 ) or
           ( in = 0.0 and output > 0.7 )
    end;

end;
```

The non-deterministic character is also expressed by redefining the predicate *fixpointp* that helps the system determine whether or not the net has converged. Here *fixpointp* expresses the non-deterministic range of the unit function modulo some tolerance because of the slow convergence of the sigmoid function close to 0 and 1.

From the superclass UNIT, this class inherits a sigmoidal squashing function with a fixed steepness (t=10, cf. Fig. 7).

A UNIT is a TWO_PHASE_UNIT, it can be used with all modes of stepping. The stepping mode for the COL4_NET is random serial. This net would also converge to a solution in synchronous parallel mode due to the random potential that is taken by *compute* above.

2.3 Learning

One of the main intended uses of ICSIM is to allow the user to experiment with different learning algorithms. The encapsulation of learning, in order to separate it as much as possible from the structure of the network, is the most important issue. The cascade correlation algorithm is chosen to illustrate the approach described, because it is one of the more complex learning algorithms, and exercises many aspects of the simulator.

Skipped psfile=ps/sigmoid.ps

Figure 7: A temperature dependent sigmoidal unit functions

After a brief description of the cascade correlation example, the constraints and criteria for implementing learning will be discussed. Then the solution chosen by ICSIM will be described, illustrated by the implementation of cascade correlation.

2.3.1 Cascade Correlation: The Algorithm

The network starts out as a perceptron, with an input layer, an (empty) hidden layer, and an output layer. The input layer is fully connected to the output layer.

The connections to the output layer are trained using some form of gradient descent - quickprop [5] is a fast second order method to achieve this. Training stops when the total error of the output layer does not change by even a small amount (the output change threshold) for a number of epochs (output patience), or when a preset maximum number of epochs is exceeded, in case the outputs do not settle but oscillate instead.

A new *candidate* node is then added to the network. It has inputs from *all* but the output nodes. To begin with, it's output is not used, except for training. The installation of this node takes place in two stages. First the inputs are trained, then it is connected to all the nodes in the output layer. All the weights coming into the output layer are then retrained.

In the first step, the inputs to the network are taught by performing gradient descent to *maximize* the correlation between the output of the candidate node, and the error of the output layer. This *correlation learning* phase is continued until the correlation does not change by more than a small amount (input change threshold) for a certain number of epochs (input patience). There is also a maximum number of epochs to handle the case where the correlation begins to oscillate wildly. Once again, quickprop may be used to perform this gradient descent.

In the second step, the new node is connected to all the outputs of the network. Its inputs are *frozen* so that they will no longer go through any form of learning. The only free (unfrozen) weights at this point, are the weights going into the output layer. These weights are then retrained using normal gradient descent (quickprop) on the outputs.

After this another unit is added and the training of the candidate units begins again. In order to improve the performance of the correlation gradient descent, not one, but several candidates are trained. However, at the end of the correlation learning phase, only the candidate with the highest correlation is tenured, and becomes a part of the network.

Note that at any given time in the algorithm, only one "layer" of weights is being retrained, so only simple (and fast) perceptron learning need be used.

The final network consists of an input layer, a cascade layer in which every succeeding node has an input from every node that came before it, and an output layer where every node has inputs from all the input and cascade nodes. See Fig 8.

Skipped psfile=ps/cascor

Figure 8: Basic Cascade Correlation architecture.

2.3.2 Design Criteria

Since learning algorithms can be very computationally expensive, the efficiency of the implemenation is important. Indirection that arises because of the structure of the design is acceptable as long as we avoid very deeply cacaded messages, and minimize overhead in tight loops. Computationally expensive parts (vector and matrix operations) should also be encapsulated, to make it easy to spawn off vector operations to specialized hardware.

Within the constraints of efficiency, the implementation should be as modular as possible, so that the learning is clearly separated from the structure of the network. This separation is necessary to facilitate independent experimentation with different network architectures and learning algorithms. ICSIM should have a library of learning methods and an independent library of network architectures, from which classes can be combined to meet the user's needs. This is only possible to the extent that the learning algorithm is independent of the network structure. Cascade correlation for instance, relies on a net that grows by adding nodes that are fully connected to all preceeding nodes.

If we view the growing net as *one* object (the structure) with a particular connectivity pattern, the cascade correlation learning method proper is independent of this structure and triggers the net to grow. On the other hand, it is often desirable to "distribute" learning and make it as local as possible. Thus the way a network is structured defines its input-output behavior (function) and at the same time how it apapts to input. To achieve modeling flexibility, ICSIM tries to be general enough to cover both these extremes. Learning may need to be implemented at various levels of the network, depending on how much must be known about other parts of the network. Ideally it would be possible to encapsulate learning at the level of the unit. With many learning algorithms this is possible. It is then easy to create any kind of network structure, simply by using the appropriate units to create the network. Sometimes, however, more global knowledge is needed, and must be encapsulated at the net level, if only for efficiency.

In order to train a network, it is sometimes necessary to control learning in the units from a higher level. This high level control can vary in complexity, from a simple two phase backprop over all patterns, to the multi-stage training of cascade correlation. We would like to encapsulate this higher level of the algorithm in a learning supervisor, so that it is encapsulated and can be re-used with different network architectures and modified easily.

We can use either a client relationship or inheritance to interact between the learning supervisor and the network i.e. the supervising class could either have the network as a client, or inherit from it. Since clients cannot break the encapsulation provided by a class, i.e. cannot use private features, a client relationship combines restriction with abstraction and is often easier to understand. An inheritance relationship, while not as well encapsulated, has faster access to the internals of the network. The inheritance relationship would tend to blur the distinction between learning and structure, since learning methods would be part of a NET class (in a descendant of the class where the structure was defined). The pure client relationship would exact a performance penalty, since all tight loops over components would now have to use the (slower) public interface to the components of a network.

ICSIM chooses a compromise. Two other classes are distinguished: a "supervisor" (which uses the client relationship) to encapsulate overall control of the algorithm, and a "learning" class (which uses inheritance) to encapsulate tight inner loops over the components.

2.3.3 Supervisor

The supervisor controls the overall behaviour of the network, which it has as a client, and describes the high level structure of the learning algorithm. It controls its client network by performing combinations of various atomic actions on it. In addition to the model, the supervisor also uses contains two other gadgets - a parameter reader and a pattern holder - to assist in the supervision of the network. The parameter holder reads in the parameters for the model (network), the pattern holder and for the supervisor itself. The pattern holder holds the training and test patterns, which are usually read in from a file. The supervisor will present these patterns sequentially to the network in the course of learning. The supervisor class is parametrized over the network model and over the pattern holder (cross-validation, for instance, is implemented by creating a new pattern holder that divides the patterns into training and test patterns, and can step through them independently).

```
class ANY_SUPERVISOR{M ,P} is
ANY;

  model: M
  pat_holder: P;
  params: PARAM_READER;

  init(init_pmap: PMAP) is ...
  – read in pmaps from an input file and set up
  – the model and pat_holder

  run is ...
  – default running behaviour.

  learn is ...
  – redefined in subclasses to reflect actual learning
  – methods.

  forward_pass is ...
  – The computation of a network.

  epoch is ...
  – Forward pass for various patterns

end;
```

The protocol that is defined for the supervisor consists of

- creation: *create, init*
- control: *run, learn*
- description: *describe*

When the supervisor is created, it first calls on the parameter reader to read in the network parameters from a parameter file, using the parameter reader. The parameters are in the form of PMAP's (associative lists of parameters and their values), which may be nested, in order to reflect the creation of nested networks. The appropriate PMAP's are then passed as the parameter to the create routines, so that the network is set up as specified by the parameter file.

After its creation (during which the model etc. are instantiated), the supervisor is sent the message run, which performs the stepping, training, testing and network modification that the algorithm requires. The supervisor controls the algorithm by passing messages (calls) to the units via the medium of the net (model) and learning classes.

The supervisor thus controls all the algorithm, by performing actions on a network, down to the level where the components must be accessed. Actions that do need to access the components directly are encapsulated in the network class in order to achieve efficiency.

The cascade correlation supervisor provides a high-level description of the cascade correlation algorithm. The different kinds of epochs, for training output and hidden nodes, are described in the supervisor.

We first declare the parameters that the supervisor itself needs. These parameters will be instantiated from the PMAPS in the parameter file during the create routine of the supervisor (in ANY_SUPERVISOR).

```
class CC_SUPERVISOR{M,P} is ANY_SUPERVISOR{M,P};
  -- ...

  output_patience: INT;
  -- ... all other parameters for learning
```

The overall algorithm trains the output layer, trains the candidates, then installs the best candidate. The stopping criterion described here is simply the number of units. More complex stopping criteria, based on the network error for instance, are easily described.

```
  learn_and_grow(init_pmap: PMAP) is
    output_patience := init_apmap.int_opt(output_patience,10);
    in_epochs: INT := init_apmap.int(input_epochs);
    out_epochs: INT := init_apmap.int(output_epochs);
    unit_counter: INT := 0;
    until (unit_counter > num_units) loop
      output_learn(out_epochs);
      input_learn(in_epochs);
      model.install_new_unit;
      unit_counter := unit_counter + 1;
    end;
  end;
```

Training the output layer is done by performing output epochs until a certain error criterion is reached. The code for testing for patience (testing whether the err has decreased significantly), is omitted. Routines like *compute_sum_sq_error* represent the boundary between the network model and the supervisor. Since these routines need to loop over individual components, they are implemented in the network.

```
  output_learn(nepochs: INT) is
    i: INT := 0; until i = nepochs loop
      output_epoch;
      model.output_layer.compute_sum_sq_error;
      err := model.output_layer.get_sum_sq_error;
      -- ... code to test for patience
      i := i + 1;
```

```
    end;
end;
```

The output epoch represents a single pass over the patterns. Each pattern in turn is presented to the output. The pattern is forward propagated through the network, after which the error is backpropagated. The notion of a "cursor" is used to step through the patterns.

```
output_epoch is
    pat_holder.first;
    model.epoch_reset;
    until pat_holder.is_done loop
        forward_pass;
        output_backward_pass;
        pat_holder.next;
    end;
    model.output_layer.self_adapt;
end;
```

The forward pass begins by resetting the model. The current input pattern is then presented to the input layer, and the model is stepped.

```
forward_pass is
    model.reset;
    model.input_layer.present_input(pat_holder.get_input_pattern);
    model.serial_step;
end;
```

The backward pass is similar, beginning by presenting the current output pattern to the output layer, and then backpropagating the error.

As may be seen, the supervisor provides a very declarative high level description of a rather complex algorithm.

2.3.4 Learning Classes

The learning classes are mainly an interface between the supervisor and the units. They are at the same level as the network and will be combined with them, but contain only procedural information. All the functionality in the learning class could easily be placed in the net class, but separating it out makes the design more modular, and also makes it possible to combine the same procedural information with different network topologies.

Sather's facility for multiple inheritance is used to separate the learning from the net classes. The actual model then consists of a class that inherits from both the appropriate learning class and the appropriate network class. This maintains the distinction between learning and structure, without any sacrifice of efficiency.

Most of the routines of the learning class take the form of loops over the components of the network, to perform various atomic actions on these components, as shown below.

```
class BP_LEARNING{DEFERRED} is
ANY_LEARNING;
  self_adapt is
    i: INT; until i = n loop
      components[i].self_adapt; i := 1 + 1
    end;
  end;
```

The learning and network classes are combined using multiple inheritance as shown in the figure.

2.3.5 Net Classes

Cascade correlation introduces several new network classes - FULL_CC_NET, CORR_OUTPUT_NET, GROWING_CASCADE_NET. FULL_CC_NET is a container net that holds the three components network - the input net, the cascade net and the output net. It creates these networks and provides a public interface to them, for the supervisor to use. Hence, the supervisor has access to the different component networks throught the public interface of this class.

```
class FULL_CC_NET is
NET1D{$NET1D{$ANY_UNIT}};

  bias : BIAS_UNIT;
    – access functions for the input, hidden and output layers.

  input_layer: INPUT_LAYER is component[0] end;

  private create_components(init_pmap: PMAP) is ...
    – Create the input, (empty) cascade, and output nets
    – and connect them
```

Though the installation of a new unit is part of the learning schedule, it alters the structure of the network, and is hence encapsulated by the network class.

```
  install_new_unit is
    cascade_layer.install_new_unit(layer_connect_spec(0));
  end;
```

GROWING_CASCADE_NET is a CASCADE_NET with additional support for training and installing candidate units into the cascade net.

```
class GROWING_CASCADE_NET{T} is
CASCADE_NET{T};
```

```
  cand_units: List{Corr_Unit};
  best_score: Real;
```

Stepping is modified to accomodate the new candidate units. The installation of the new candidate units is performed by converting one of the candidate units into a member of the list of cascaded component units.

The Corr_Output_Net illustrates the use of multiple inheritance to denote the backprop style of learning on a Net_Layer. In addition, it needs to be able to keep error statistics for the output nodes to be used when training the candidates.

```
class Corr_Output_Net is
Net_Layer{Corr_Output_Unit};
Bp_Learning{Check};

  sum_sq_error: Real;

  compute_sum_sq_error is
    -- store the sum of the errors
    -- at each component in the sum_sq_error
    i: Int := 0; until i = components.size loop
      sum_sq_error := sum_sq_error +
      components[i].get_sum_sq_error;
      i := i + 1;
    end;
  end; end;
```

In ICSIM, the Net classes encapsulate the topology of the networks. In addition, they also serve as to communicate between the learning supervisor, and the learning methods that are at the level of the unit. Hence, they are, in a sense, the heart of ICSIM.

2.3.6 Unit Classes

Many learning algorithms can be implemented in the unit classes alone. This is to be expected, since the unit should be the basic computational structure. However, learning strategies frequently do not limit their computation to within the unit and rely on more global computation as well. As pointed out above, sometimes computation can be expressed in either a local or a global form.

The cascade correlation example defines three main classes of units - the Qp_Unit, the Corr_Unit and the Corr_Output_Unit.

Qp_Unit implements the quickprop algorithm [5], by inheriting from backprop and modifying the weight update method. The create routine must also be modified in order to take into account the extra parameters and arrays that the quickprop unit requires. The Qp_Unit is not used directly, but both the Corr_Output_Unit and the Corr_Unit inherit from it.

The Corr_Output_Unit is used in the construction of the output layer. It keeps track of the sum of the error it has seen and the sum of the error squared. This is accumulated over an epoch before the candidate units are trained, since the candidate units need to know this information in order to compute their own error.

```
class CORR_OUTPUT_UNIT is
QP_UNIT;

  sum_error: REAL;
  sum_sq_error: REAL;

  compute_current_error is
    local_error := d_unit_fn(output, 0.0) * feedback_sum;
    sum_error := sum_error + local_error;
    sum_sq_error := sum_sq_error + local_error * local_error;
  end;
```

The CORR_UNIT is used for candidate units. This unit has a pointer to the output network, so as to get access to the error statistics of the output nodes, which are CORR_OUTPUT_UNITS. It needs these in order to compute its own correlation error.

```
class CORR_UNIT is
QP_UNIT;

  corr: LIST{REAL}; – Correlations with output units
  prev_corr: LIST{REAL}; – Previous correlations
  sum_value: REAL; – Sum of values over an epoch

  output_corr_net: CORR_OUTPUT_NET;
    – points to the output network

  compute_correlations is
    sum_value := sum_value + output;
    o: INT := 0; until o = output_corr_net.size loop
      corr[o] := corr[o] +
      output_corr_net.component(o).get_local_error * output;
      o := o + 1;
    end;
  end; end;
```

With the exception of the CORR_UNIT, the UNIT level implementations of the algorithms are completely local. Any different network structure may use quickprop, for instance, by using QP_UNITS to construct the network. Even in cascade correlation, if we change the CORR_UNIT to inherit from BP_UNIT instead of QP_UNIT, we can use backprop instead of quickprop.

2.4 Views

From the user's perspective, views are the first-level objects for presenting models and interact with them. Views have a *passive* character to the extent that they are the medium through which models are seen. At the same time they have the *active* character of an instrument to the extent that they allow the user to manipulate models.

Each view encapsulates a specific technique of presentation and to this end a moderate amount of knowledge about the object being presented. Since views are the only medium to 'see' and 'manipulate' models, it is natural in an object-oriented setting to associate view-related functionality with the model objects themselves and to localize the information and logic needed for the purpose of presentation and interaction. On the other hand, in the presence of many parallel objects, we do not want to duplicate the storage needed for the presentation and interaction logic and do not want to distribute this logic more than necessary to the objects executing in parallel. Hence there is a tradeoff between object-specific functionality local to the objects and central functionality for managing presentations. This tradeoff is particularly evident when parallel simulations are considered in which models execute on parallel processors while the user monitors the simulation at the local workstation.

Our compromise in ICSIM is to have different kinds of views that fall in different ranges between these two extremes. So far we distinguish five kinds of views. The most general ones, not restricted to our application, are the *text interaction view* and *tour view*. These views maintain the 'what-you-see-is-the-object' metaphor, and are indeed associated to the objects themselves.

The text interaction view, or *text view* gives a textual presentation of the current state of the simulation (cf. below) and lets the user enter in a textual dialog. It is based on various *describe* routines and allows interaction with a single top-level net in terms of textual menus. The menus offer the main simulation functions of the corresponding net class. By inheritance, all user-defined objects will 'know' how to describe themselves. This eases rapid prototyping of models. In time, the user will typically redefine parts of this functionality to adopt net descriptions to the specific problem at hand.

The following is an extract of the dialog that results from the four-color classes we have described above.

```
Text View (choose command):
1: New
2: Reset
3: Sync Bound
4: Micro Step
5: Step
6: Step size
7: Solve (Care!)
8: Fixpoint?
9: Describe
10: View
11: QUIT
Choose (Default: New): step si
=> Step size
Typein number of steps (Default: 1): 10
Choose (Default: Step size): ste
=> Step
Region=0   G : P=-87.4225,O=0;P=-102.844,O=0;P=30.6219,O=1;P=-20.6846,O=0
Region=1 B   : P=41.5084,O=1;P=-112.997,O=0;P=-117.051,O=0;P=-21.3759,O=0
Region=2   G : P=-113.033,O=0;P=-29.1242,O=0;P=40.4869,O=1;P=-107.736,O=0
```

```
Region=3  R  : P=-80.5654,O=0;P=95.0193,O=1;P=-112.285,O=0;P=-55.5378,O=0
Region=4    W: P=-42.0739,O=0;P=-22.7232,O=0;P=-44.9267,O=0;P=15.5342,O=1
Region=5  R  : P=-82.594,O=0;P=57.0895,O=1;P=-114.38,O=0;P=-92.1095,O=0
Region=6    W: P=-28.485,O=0;P=-62.5832,O=0;P=-156.564,O=0;P=63.5484,O=1
Region=7   G : P=-69.826,O=0;P=-123.127,O=0;P=22.5425,O=1;P=-40.0455,O=0
Region=8   G : P=-95.7435,O=0;P=-49.759,O=0;P=43.1937,O=1;P=-102.074,O=0
Region=9 B   : P=33.9479,O=1;P=-86.9696,O=0;P=-110.278,O=0;P=-93.9951,O=0
Choose (Default: Step): solve
=> Solve (Care!)
Fixpoint reached...
Region=0   G : P=-108.461,O=0;P=-89.0467,O=0;P=96.153,O=1;P=-36.0983,O=0
Region=1 B   : P=25.4792,O=1;P=-49.5313,O=0;P=-134.411,O=0;P=-55.9796,O=0
Region=2   G : P=-47.1786,O=0;P=-65.5438,O=0;P=43.6305,O=1;P=-79.8739,O=0
Region=3  R  : P=-73.183,O=0;P=60.8683,O=1;P=-97.7836,O=0;P=-68.6225,O=0
Region=4   G : P=-34.2809,O=0;P=-50.2365,O=0;P=28.1771,O=1;P=-64.8654,O=0
Region=5  R  : P=-91.8668,O=0;P=47.9891,O=1;P=-53.9801,O=0;P=-116.018,O=0
Region=6    W: P=-138.609,O=0;P=-40.5245,O=0;P=-55.539,O=0;P=94.1444,O=1
Region=7 B   : P=95.265,O=1;P=-137.771,O=0;P=-140.15,O=0;P=-118.357,O=0
Region=8   G : P=-104.173,O=0;P=-26.8236,O=0;P=37.5878,O=1;P=-82.4699,O=0
Region=9    W: P=-96.8263,O=0;P=-89.8717,O=0;P=-93.9958,O=0;P=89.2666,O=1
```

The *tour view* allows to tour and navigate through the 'land-of-instances'. It is useful in debugging, including high-level model debugging to understand the average and limit cases of the model behavior that occurs during the simulation. In the tour view, attributes can be inspected, objects can be created on the fly, and exported routines can be executed on them. Most classes inherit general, although inefficient, exported routines for reaching related objects directly or describing the object in the current focus (cf. Fig. 9).

The tour view is currently implemented by exploiting the Sather debugging environment *SDB*. SDB offers interative inspection, stepping, tracing and breakpoint facilities under Emacs. Under X the tool supports a simple point-and-click interface on textual, i.e. source-level, descriptions of objects.

The above textual views, including the tour view, do not add attributes to instances. They are only based on routines and runtime information about the classes and the features they support. In general, however, views may require instance-level information. In particular *graphic views* described below are therefore separate from model classes in ICSIM. They are encapsulated in terms of more or less model independent classes and support different metaphors for visualization. In our current, still floating user interface design, we call these views *block view*, *map view* and *profile view*.

The *block view* is loosely related to the tour view. It presents the structure and interconnection of a net at different levels of the hierarchy in a symbolic way by nodes and edges. This view is useful at a high level of abstraction where a single edge between two net nodes represents a complex interconnection pattern. Traversals on these graphs and blowing up its details is comparable to a restricted tour through the corresponding objects. Nets can be instantiated from a palette of existing classes and the choice of an edge corresponds to the invocation of a specific connection routine. Due to the symbolic level it provides, the block view is a kind of 'point-and-click' entry point for ICSIM beginners.

The *map view* maps net state(s) to a geometrical layout. For instance, the units of a layered net can be mapped to a two-dimensional cellular plane showing the activation in limited regions of a net through colored cells on the plane. Typically the arrangement

```
+-----------------------------------------------------------------------+
|            +----------------------------------+ +--------------------+|
|            |  Object: #12020C  Class: COL4UNIT| |A: show Attributes  ||
|            |  Attributes: 4  Routines: 164    | |B: Back up          ||
|            +----------------------------------+ |C: Create object    ||
|(1)   inputs: ARRAY_SET -->                      |E: Execute routine  ||
|(2)   weights: ARRAY_SET -->                     |F: Forget object    ||
|(3)   potential: REAL = -24.993568               |G: Goto object      ||
|(4)   output: REAL = 0.000000                    |I: check Invariant  ||
|                                                 |L: List objects     ||
|                                                 |N: Name object      ||
|                                                 |Q: Quit             ||
|                                                 |R: show Routines    ||
|                                                 |S: Set attribute    ||
|                                                 |W: Wash screen      ||
|                                                 |X: set-up eXecution ||
|                                                 +--------------------+|
+-----------------------------------------------------------------------+
|Your command:                                                          |
+-----------------------------------------------------------------------+
```

Figure 9: Tour view: based on the Sather Source-Level Debugger

of cells is very regular, i.e. cells are uniformly distributed in the two dimensions (*array metaphor*).

Finally, the *profile view* presents the surface of some function of the net state or of the product of state and time. Histograms, plots and error surfaces are examples of this metaphor.

To simplify the application and 'programming' of views, views can be connected to nets much like nets are interconnected in the model world. As a by-product of this design, storage for view related information (attributes, instances) is dynamically allocated only when needed. And views can then be saved to persistent storage like nets if the user wishes to save them. Moreover many views can be 'open' on the same object at any time. And the state or behavior of many objects can be viewed with a single view. In other words, there is an N-to-M relation between models and views.

In order to meet the requirement for encapsulation, type-specific views can inherit from the respective net classes. In this way, views that are meaningful only for certain types of nets can encapsulate model specific information without having to include presentation information in each model instance. This approach seems to be a reasonable compromise with respect to the tradeoff between object-specific presentations (controlled by the model object) and presentation objects that sense and control models and, in this sense, centralize user interface functionality.

3 Conclusion

We described the design of ICSIM, a simulator for connectionist nets. The choice of an object-oriented approach to the problem is promising and seems to have scope also in the direction of parallel simulations.

The design of ICSIM is centered around nets instead of units. The decomposition of large nets into subnets allows to choose adequate data structures without compromising the general simulation-oriented functionality. At the same time nets allow a user to choose the granularity of distributed representation and parallelism without the need of homogeneous (SIMD type of) representation where this would be unnatural.

Acknowledgement

We are grateful to Jerry Feldman with whom we had many stimulating discussions on the design of ICSIM. Also thanks to Susan Weber, Franz Kurfess and Richard Durbin who worked with prototype versions of the similulator and had many useful suggestions for improvements. Also thanks to Jeff Bilmes with whom the first author worked on the design of the user interface.

The details of the cascade-correlation algorithm are based on Scott Crowder's C implementation.

References

[1] Alexander, I (ed.): *Neural Computing Architectures: The design of brain-like machines.* Cambridge: MIT Press, 1989

[2] Coss, R.G., and Perkel, D.H.: The function of dendritic spines: a review of theoretical issues. *Behavioural and Neural Biology,* **44,** pp. 151-185 (1985)

[3] D'Autrechy, C.L. et al: A general-purpose simulation environment for develping connectionist models. *Simulation* **51***,* 1, pp. 5-19

[4] Fahlman, S. and Lebiere, C. *"The Cascade-Correlation Learning Architecture".* Carnegie-Mellon Report CMU-CS-90-100, 1990

[5] Fahlman, S. "Faster-Learning Variations on Back-Propagation: An Empirical Study". In: *Proc. of the 1988 Connectionist Models Summer School,* Morgan Kaufmann, 1988

[6] Feldman, J.A. and Ballard, D.H.: Connectionist models and their properties. *Cognitive Science,* 6, pp. 205-254

[7] Feldman, J.A. et al: Computing with Structured Connectionist Networks, *CACM* **31***,* 2, pp. 170–187, (1988)

[8] Goddard, N.: *The Rochester Connectionist Simulator: User Manual,* TR, Univ. Rochester, 1987

[9] Hecht-Nielson, R.: Neurocomputing: Picking the Human Brain. *IEEE Spectrum, March 1988,* pp. 36–41

[10] Korb, T., Zell, A.: A declarative neural network decription language. *Microprocessing and Microprogramming* **27***,* North-Holland, pp. 181–188 (1989)

[11] Lange, T.E. et al.: *DESCARTES: Development Environment for Simulating Hybrid Connectionist Architectures.* TR UCLA-AI-89-16, Los Angelos: UCLA, 1989

[12] McClelland, J. L., Rumelhart, D. E., and the PDP research group: *Parallel distributed processing: Foundations.* Cambridge, MA: Bradford Books, 1986

[13] Omohundro, S.: *The Sather Language.* TR, Berkeley: ICSI, to appear, 1990.

[14] Shastri, L. and Ajjanagadde, V.: *From associations to systematic reasoning.* TR, Philadelphia: Univ. of Pennsylvania, 1989

[15] Waltz, D., Feldman J.A. (eds): *Connectionist models and their implications: readings from cognitive science.* Norwood, N.J.: Ablex Pub. Corp., 1988.

[16] Wassermann, P.D.: *Neural Computing: Theory and Praxis.* New York: Van Nostrand Reinhold, 1989

[17] Wilson, M.A. et al.: *Genesis: A system for simulating neural networks.* Proc. of '89 NIPS conf., also TR: Pasadena: Cal. Inst. of Tech., 1989

[18] Wilson, M.A. et al.: *Genesis&XODUS: General Purpose Neural Network Simulation Tool.* Proc. of '89 USENIX conf., also TR: Pasadena: Cal. Inst. of Tech., 1989