# ICSIM: Initial Design of An Object-Oriented Net Simulator

Heinz W. Schmidt*

TR-90-055
October, 1990

**Abstract**

ICSIM is a connectionist net simulator being developed at ICSI. It is object-oriented to meet the requirements for flexibility and reuse of models 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 routines.

The report gives an overview of the simulator. The class structure and some important design decisions are sketched and a number of example nets are used to illustrate how net structure, connectivity and behavior are defined.

*ICSI, Berkeley, California; on leave from: Inst. f. Systemtechnik, GMD, Germany

# 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 and acceptable cost.

Flexibility is an 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. [10, 13, 7, 1, 14]). Different simulators serve different purposes ranging from modeling bio-chemical processes in the human brain (e.g. [15]) to developing structured connectionist models of artificial memory, recognition and reasoning processes (e.g. [6, 9, 3]).

Efficiency is equally important; the simulation of the conceptually massively parallel nets, for instance in real-time speech recognition, may take hours on sequential machines.

Existing simulators like the Rochester simulator [6] or Genesis [15] 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 that allows the user to change the representation and/or structure of nets during a simulation. This is particularly important in case of long simulation runs. Otherwise, in a non-incremental environment, these long runs tend to repeat – in a different branch of computer science – the problems of the batch-oriented software development style of the early seventies with their long turn-around time in a slow 'edit-compile-run-debug' cycle. For instance, in the development of the Genesis simulator this need was specifically addressed by an intermediate shell language interpreter [16] 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 support command-language-like interfaces in an incremental prototyping environment but also to integrate separate tools driving the simulator in a less incremental fashion as a back-end via procedure call interfaces. 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. [8, 3]).

In the short range future, we expect network sizes in the range of some hun-

dred 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 some novel artificial neural network concepts, especially: modular architectures, shared structures (e.g. shared weights), and learning.

2. Provide simple means to extend and/or add types of neural networks and types of units.

3. Permit incremental (during simulation) and non-incremental construction of networks, size change and restructuring of networks.

As a means 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 design is on the structure of the class libraries and to some extent the separation and interaction between the simulator proper and the user interface objects. The implementation is written in Eiffel and a toy simulator is written in Common Lisp enabling us to experiment with new ideas in an interpretative environment.

The design tries to assist the specialist scientific community in teaching and research in artificial neural networks. We envisage three types of 'interfaces' to ICSIM that require 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.

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

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 'single-point' redefinition. If this is done well, most users should be able to do their simulations at this level.

3. An *advanced library* with a set of dedicated classes is to support the skilled object-oriented programmer and/or experienced ICSIM user to build new prototypes 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, neuron models 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, many views may filter different aspects of a single chosen model.

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 processors have local memory to hold model instances combined with a user interface running on a workstation that holds view instances.

## 2.1 Excursion: Eiffel Classes

The examples we present in the course of the following sections are written in Eiffel. We shortly sketch the constructs we use in this text.

Eiffel is a statically typed language. In Eiffel every data type is a class. Besides introducing a type for typing variables and arguments or results of operations, a class supplies

1. dynamic instantiation; it is a template for instantiating a compound data object with a number of *attributes* – or instance variables; *create* is a distinguished procedure that instantiates an object and initializes it; usually class bodies contain a create procedure describing the user-defined initialization.

2. an encapsulated implementation of an abstract data type. The abstraction or interface of this encapsulation is given by naming a set of *exported* operations; instances are protected in that calls from different instances must go through these exports; the operations of the class are called *features* in Eiffel; they include attributes and *routines* which are either *functions* or *procedures.*

3. multiple inheritance; classes can be combined to form new classes; exported and private features are inherited; this is one of the central concepts of object-oriented programming; Eiffel programmers speak of *descendents* and *ancestors* of classes in this context.

4. subtypes; in the type system, inheritance gives rise to subtyping; for instance if class BP_LAYER inherits from class NET, it is also a subtype of NET. If $x$ has type BP_LAYER, then $x$ can be passed as argument or assigned to variable that accepts a NET; loosely speaking we can consider a subtype a subset; this means the set of all conceivable NET objects consists of all NET instances but also of the instances of the subclasses of class NET. In other words the variable $x$ can has a static type, here NET and its value at runtime has a dynamic type, here BP_LAYER that is a subtype of the static type.

5. overloading and dynamic binding: when forming new classes with inheritance, usually a few features are redefined to achieve a slightly modified behavior, or they are renamed to adapt the terminology to the purpose of the newly formed type or to make an inherited feature available beside the new one. A redefinition of a an inherited feature makes the name overloaded. For one class of object it has a different meaning and implementation than for another one. Dynamically, i.e. is at runtime this is achieved by late binding: the class specific code is dispatched to according to the dynamic type of the object receiving the call. Given a feature call *x.f(a,b)*, only at runtime is the proper feature (code) bound to the name *f* by looking up this dynamic type.

6. incremental compilation; a class is a module; the compiler keeps track of changes and runs only the necessary passes on a subset of classes refered to from the class being compiled.

Here is a typical class header:

**class** BP_LAYER [U → BP_UNIT]

**export repeat** NET

**inherit** NET [U] **redefine** step, create_unit, self_adapt;

**feature**

...

**end**

The first declares the class BP_LAYER (short for: backpropagation layer). This class is parameterized, the formal parameter, listed between square brackets, is U. The arrow notation indicates that an actual parameter substituted for U must be a descendent of class BP_UNIT.

The class name is followed by the export list. Here the keyword *repeat* just states that this class exports whatever the class NET exports.

The next line lists the ancestors, i.e. the superclasses, in this case a single class NET with actual parameter U suggesting that a BP_LAYER composed of units of type U is a NET composed of units of type U. Subsequently the names of features are listed that are going to be redefined in the class body.

The keyword *feature* is the beginning of the class body and is followed by attribute and routine definition.

The dots in line 5 do not belong to the Eiffel syntax. We will sometimes use ellipsis to skip over irrelevant details.

The feature definitions in the class bodies are more or less self-explaining, maybe except for:

1. dot notation is used to distinguish the argument whose dynamic type determines the feature that implements the call. Multiply dotted expressions are allowed, i.e. if the last feature called is a function that returns an object we can apply another feature to it and so on. For instance in *x.b.step(5)*, we assume that the value of some variable *x* is an object that supports a feature *b*. *b* must return another object which supports a feature *step* that takes one argument.

2. in some examples we use the Eiffel 'inspect' construct, which is a case statement allowing a conditional branch according to either character or integer values.

3. in another example we use so-called 'once'-functions. Effectively, such a function is a shared variable. It is only executed with the first call, later calls 'read' the result of the first call.

## 2.2 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.

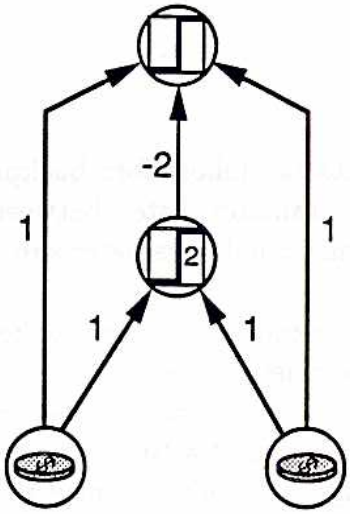


Figure 1: Xor net

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

Upon creation the interactive text view class creates, initializes and then runs the interactive view by calling the superclass initialization routine. This routine in turn calls back *create_net* redefined in the class body to return a net of the proper type. All the simulator behavior, i.e. command prompting and interpretation, net access and so forth are inherited from the superclass TEXT_VIEW_1NET and its ancestors. The main purpose of the class XOR_SIM is to select the proper type of view and to specialize the type of net to be simulated.

```
class XOR_SIM inherit TEXT_VIEW_1NET [XORNET] redefine create_net
feature

    create is do text_view_1net_init end ;

    create_net: XORNET is do Result.create end ;

end
```

An (XORNET) 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 an inherited initialization routines and then wires them. For simplicity of the example we use integers here to 'name' the units. Unit 0 and 1 represent the input units, unit 2 the hidden and unit 3 the output unit. The initialization routine calls back the routine *create_unit* redefined to 'declare' the proper type of units. Instances of class BOOL_RND_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. *con1* and *con2* are connection specification objects[2] further detailing the properties of the interconnection structure. In this example they are constant connections (CONST_CON) selecting the weights depicted in Fig. 1 (1.0 and −2.0).

```
class XORNET repeat export NET
inherit NET [ANY_UNIT] redefine step, create_unit

feature

  create is
    do
      net_init (4); - - create 4 units and then connect them one by one
      unit(2).connect (unit(0),con1);
      unit(2).connect (unit(1),con1);
      unit(3).connect (unit(0),con1);
      unit(3).connect (unit(1),con1);
      unit(3).connect (unit(2),con2);
    end ;

  create_unit (i: INTEGER): ANY_UNIT is
    local in: BOOL_RND_UNIT; h: VAR_BOOL_UNIT; o: BOOL_UNIT;
    do
      inspect i
      when 0..1 then in.create ; Result:=in;
      when 2 then h.create (2,2.0); Result:=h; - - 2 inputs, threshold=2.0
      when 3 then o.create (3); Result:=o;
      end ;
    end ;
```

[2]These are explained in more detail in a subsequent section.

```
    step is do serial_step end ;

    con1: CONST_CON is once Result.create (1.0,sys.always) end ; -- weight 1
    con2: CONST_CON is once Result.create (-2.0,sys.always) end ; -- weight -2

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_SIM, is compiled.

## 2.3 Models

Neural 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.

### 2.3.1 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

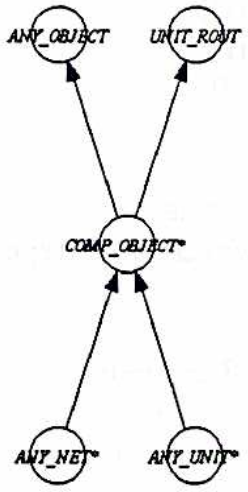


Figure 2: Comp_object: Common functionality of nets and units

in the class COMP_OBJECT, short for *computation object*, depicted in Fig. 2. This class defines the protocol for interconnecting and triggering the computation steps

of nets and units. As shown in the figure, COMP_OBJECT inherits from ANY_OBJECT and UNIT_ROUT. ANY_OBJECT 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 debugging. Class UNIT_ROUT packages various unit functions like sigmoids and thresholds so that they are available in the context of all unit and net classes.

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, however, 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 [5, 12, 9], 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. The most general such net class is ANY_NET (cf. Fig. 2). It defines the minimal protocol that all nets must observe.

### 2.3.2 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 [4, 12], 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. 3.

For the class ANY_UNIT, this value is computed but not stored. In other classes

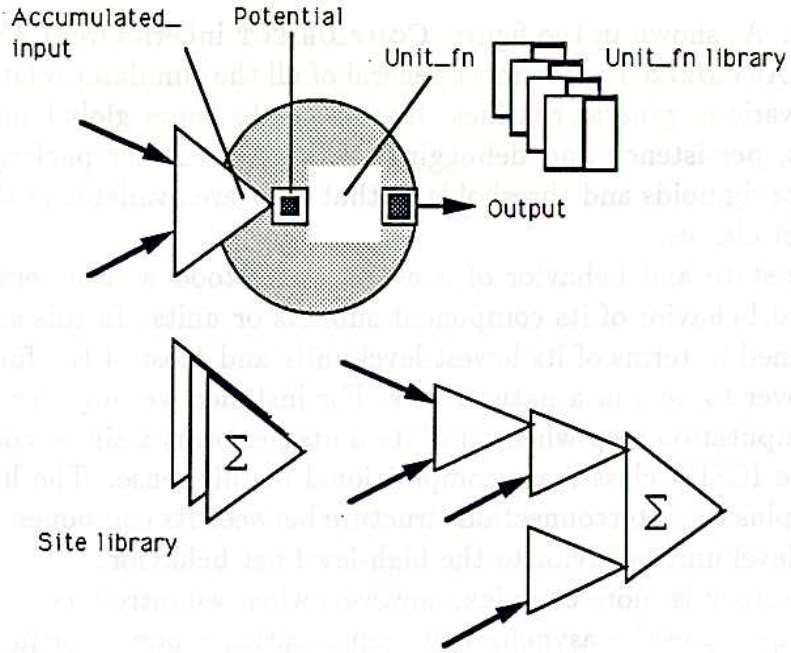


Figure 3: Unit synthesis

(PHASE2_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, all of them will change their output (cf. Section 2.3.4 below).

### 2.3.3 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 training, specific unit classes embody learning 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 OUTPUT_OBJECT 4. 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. 3 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[3]. 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. In order to connect nets in a flexible and abstract way, ICSIM includes ob-

[3]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].

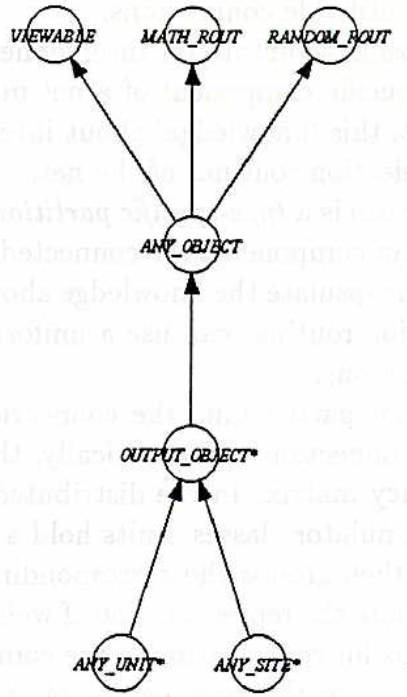


Figure 4: Output_object: Common functionality of units and sites

jects encapsulating *connection specifications* that parametrize the various connection procedures. 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. 11). A connection specification can 'tell' *x_connect* to skip certain connections, to connect with some probability only and so on. It also 'knows' the initial weight to use and for multi-site units it knows which site to connect to.

In this way users can customize interconnection structure on a high level – a more 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, the most general connection specification, called ANY_CON, supports the routines (Fig. 5) *connectp*, *which_site* and *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

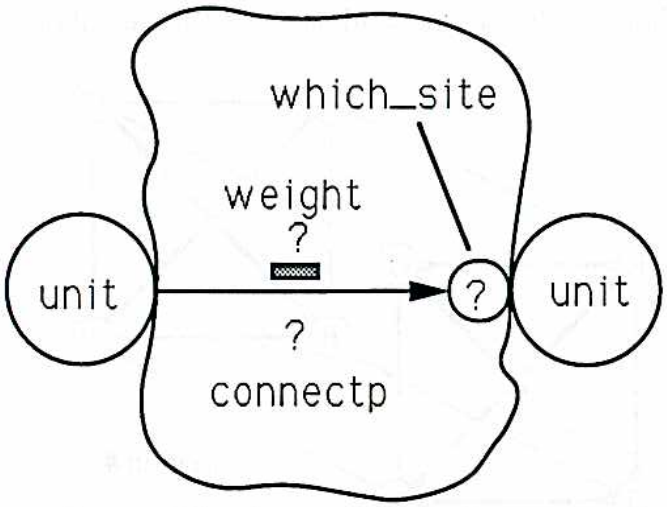


Figure 5: Connection specifications customize interconnection structure

type of connection (site) is built and will, subsequently, choose the next site for all units.

The following example shows hierarchical nets and the use of different connection primitives. Our problem consists of coloring the map shown in Fig. 6, such that

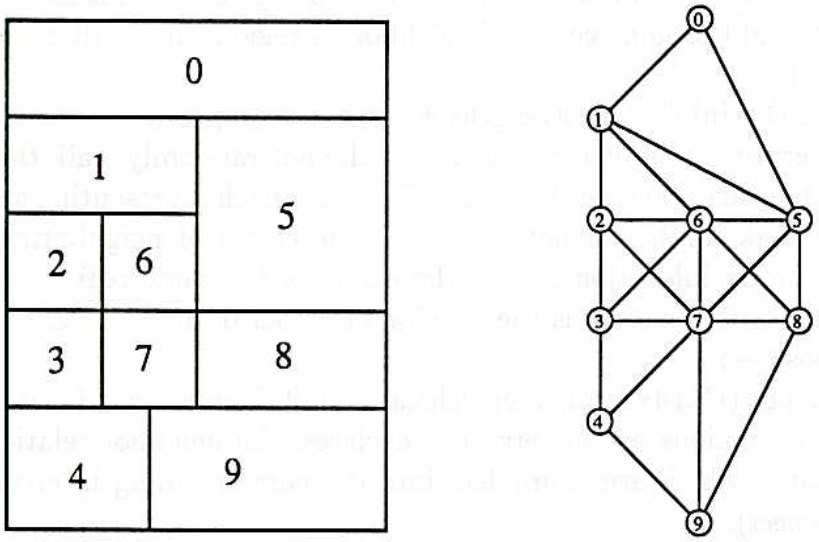


Figure 6: Region map and neighbor graph

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

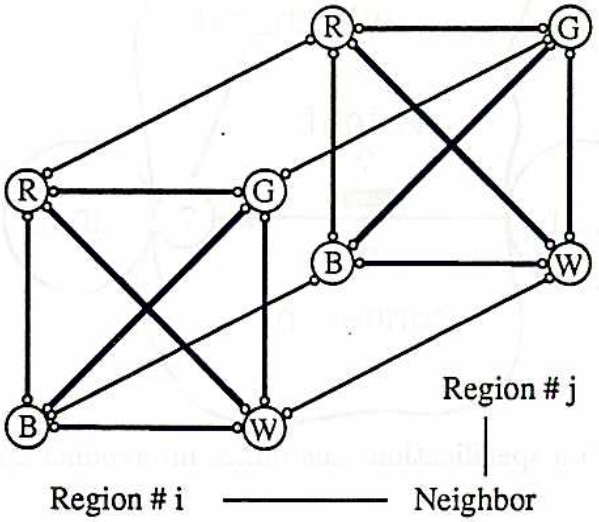


Figure 7: Interconnection structure

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. 7, 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 (here $-1$) and a strong inhibition 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. Here we choose $-5$.

The four-color net (COL4NET) is hierarchically built from regions (COL4REG). A COL4NET 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 COL4UNITS. It creates its four color units as part of its own creation and interconnects them internally.

```
class COL4NET inherit HIERARC_NET [COL4REG] redefine create_subnet
feature

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

    create is
      do
        hierarc_net_init (10);              - - create 10 regions
        neighbor (0,1); neighbor (0,5);   - - declare neighbor relation
        neighbor (1,2); neighbor (1,6); neighbor (1,5);
      ...
      end ;

    create_subnet (i: INTEGER): COL4REG is
      - - called as part of the hierarc_net creation protocol.
      do Result.create (i) end ;

end


class COL4REG inherit NET [COL4UNIT] redefine create_unit
feature

    country: INTEGER; - - remember name for user interface

    create (c: INTEGER) is
      do
        country:=c;
        net_init (4); - - create and initialize four color units
        complete_connect (strong_inhibit); - - strong mutual exclusion
      end ; 4

    create_unit (i: INTEGER): COL4UNIT is
      - - called back by net creation; ignore the unit name i
      do Result.create end ;
    ...
end
```

The connections specifications *weak_inhibit* and *strong_inhibit* are constant connection specifications like in the previous (Xor) example. Together with other constants, they are defined in a small class COL4OBJ from which all the above inherit.

```
strong_inhibition: REAL is -5.0;
strong_inhibit: CONST_CON is
  once Result.create (strong_inhibition,sys.always) end ;

weak_inhibition: REAL is -1.0;
weak_inhibit: CONST_CON is
  once Result.create (weak_inhibition,sys.always) end ;
```

### 2.3.4 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 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 *steps* 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 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 $\infty$ 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 composed of the unit state $x_i$. 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 state. For a unit the predicate *fixpointp* represents the local fixpoint condition $x_i = f_i(\mathbf{x})$.

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 PHASE2_UNIT does. The user may design subclasses of ANY_UNIT instead of subclasses of PHASE2_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 (COL4UNIT) 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 COL4UNIT
inherit UNIT redefine fixpointp, compute; ...

feature

    compute is
      do potential:=unif_rnd - accumulated_input/strong_inhibition + 0.5 end ;

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

    create is do unit_init (4) 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.

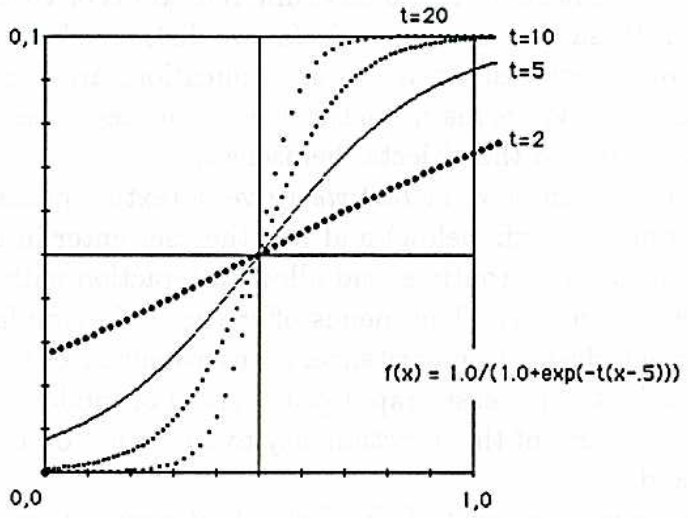


Figure 8: A temperature dependent sigmoidal unit functions

From the superclass UNIT, this class inherits a sigmoidal squashing function with a fixed steepness (t=10, cf. Fig. 8).

A UNIT is a PHASE2_UNIT, it can be used with all modes of stepping. The stepping mode for the COL4NET 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.

## 2.4 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=-87.4225,0=0;P=-102.844,0=0;P=30.6219,0=1;P=-20.6846,0=0
Region=1 B   : P=41.5084,0=1;P=-112.997,0=0;P=-117.051,0=0;P=-21.3759,0=0
Region=2   G : P=-113.033,0=0;P=-29.1242,0=0;P=40.4869,0=1;P=-107.736,0=0
Region=3  R  : P=-80.5654,0=0;P=95.0193,0=1;P=-112.285,0=0;P=-55.5378,0=0
Region=4    W: P=-42.0739,0=0;P=-22.7232,0=0;P=-44.9267,0=0;P=15.5342,0=1
Region=5  R  : P=-82.594,0=0;P=57.0895,0=1;P=-114.38,0=0;P=-92.1095,0=0
Region=6    W: P=-28.485,0=0;P=-62.5832,0=0;P=-156.564,0=0;P=63.5484,0=1
Region=7   G : P=-69.826,0=0;P=-123.127,0=0;P=22.5425,0=1;P=-40.0455,0=0
Region=8   G : P=-95.7435,0=0;P=-49.759,0=0;P=43.1937,0=1;P=-102.074,0=0
Region=9 B   : P=33.9479,0=1;P=-86.9696,0=0;P=-110.278,0=0;P=-93.9951,0=0
```

```
Choose (Default: Step): solve
=> Solve (Care!)
Fixpoint reached...
Region=0   G : P=-108.461,0=0;P=-89.0467,0=0;P=96.153,0=1;P=-36.0983,0=0
Region=1 B   : P=25.4792,0=1;P=-49.5313,0=0;P=-134.411,0=0;P=-55.9796,0=0
Region=2   G : P=-47.1786,0=0;P=-65.5438,0=0;P=43.6305,0=1;P=-79.8739,0=0
Region=3  R  : P=-73.183,0=0;P=60.8683,0=1;P=-97.7836,0=0;P=-68.6225,0=0
Region=4   G : P=-34.2809,0=0;P=-50.2365,0=0;P=28.1771,0=1;P=-64.8654,0=0
Region=5  R  : P=-91.8668,0=0;P=47.9891,0=1;P=-53.9801,0=0;P=-116.018,0=0
Region=6    W: P=-138.609,0=0;P=-40.5245,0=0;P=-55.539,0=0;P=94.1444,0=1
Region=7 B   : P=95.265,0=1;P=-137.771,0=0;P=-140.15,0=0;P=-118.357,0=0
Region=8   G : P=-104.173,0=0;P=-26.8236,0=0;P=37.5878,0=1;P=-82.4699,0=0
Region=9    W: P=-96.8263,0=0;P=-89.8717,0=0;P=-93.9958,0=0;P=89.2666,0=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 an interface into the Eiffel test environment. This interface can be optionally associated to the most general class ANY_OBJECT.

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

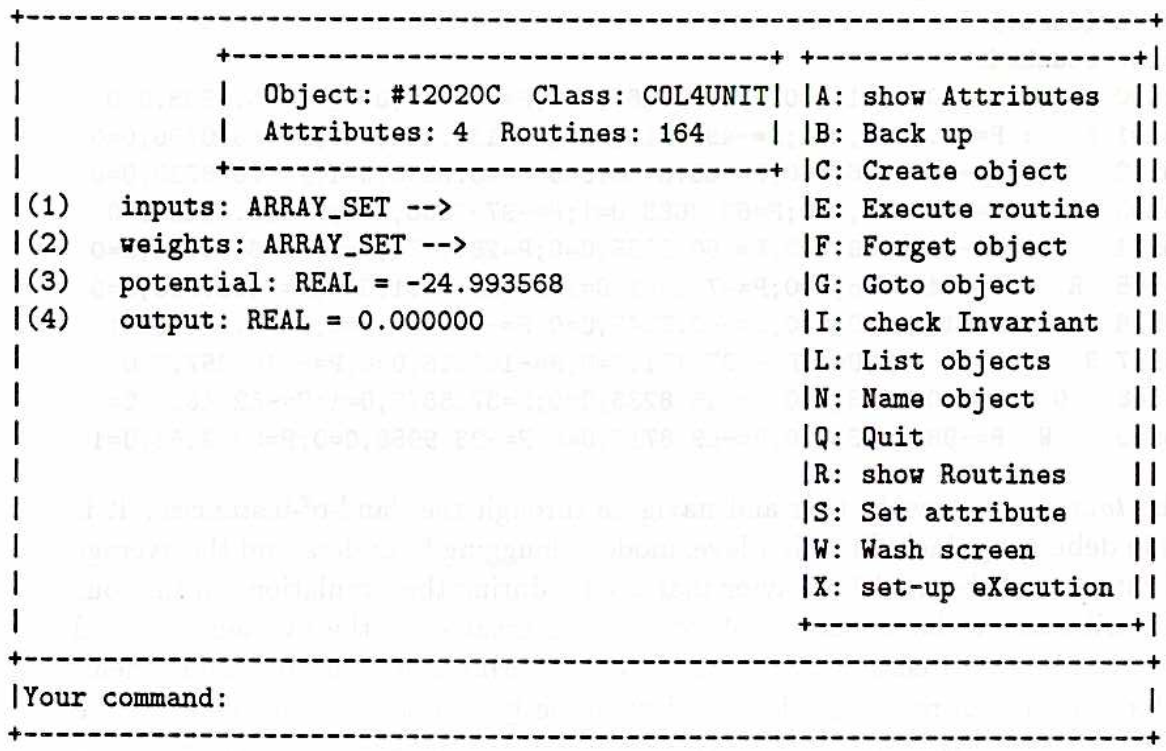


Figure 9: Tour view: based on the Eiffel test environment

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.

In our initial implementation of ICSIM we only implemented the text and tour view. Some first steps have, however, been made in the direction of graphic views

using the C++ based InterViews library.

# 3 Class Hierarchy

The three major model-related class families are nets, units and sites with their minimal elements ANY_NET, ANY_UNIT, and ANY_SITE, respectively. These classes define the minimal protocol (abstract data type) that all members of the family adhere to. They do not make any commitment with respect to the data structures used to represent any of the classes. Some of their functionality is deferred, i.e. unimplemented. Implemented functionality is based on representation-independent exported routines and hence usually inefficient. Nevertheless these implementations simplify prototyping. They provide a simple abstraction, meaningful default behavior, and representation-independent access for the user interface views.

## 3.1 Nets

Figure 10 shows part of the inheritance graph of one-dimensional nets – components that are accessed like array elements. Most of these classes are parameterized with the type of the components. For instance, NET1D [C → COMP_OBJECT], has components which adhere at least to the protocol of the class COMP_OBJECT. While nets of type NET1D can be composed of units and nets, NET instances are composed only of units.

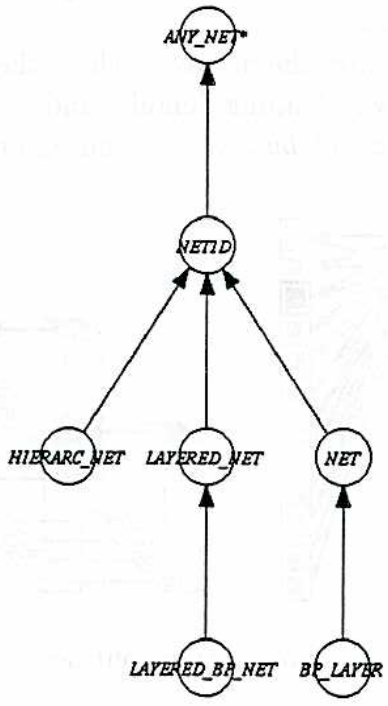


Figure 10: Net classes

The major routines supported by all nets are:

1. instantiation: *create_component*;
2. connection: *bus_connect*, *x_connect*, *complete_connect*, *disconnect*;
3. computation: *step*, *micro_step*, *reset*, *shake*, *to_fixpoint*;
4. learning: *self_adapt*;
5. visualization: *describe*, *view*.

Objects can be dynamically created and all 'container'-like objects such as nets are dynamically extensible in ICSIM. The create routines of nets are typically parameterized with the initial size of the net. The corresponding number of components is automatically created by calling back *create_component* which is usually redefined by descendent classes. Moreover, the internal connection structure is built as part of the creation procedure. For instance, a LAYERED_NET builds a cross-connection structure from each layer to the subsequent layer, from input to output layer. Additional internal connection structure and the connections with the environment have to be built by the clients of this class. Create procedures are not inherited. To allow simple modifications of the create protocol, we have followed a programming style in which each class has a globally non-generic initialization procedure. This procedure is invoked by create and inherited by descendent classes. Thus, descendents can compose their own creation behavior partly reusing the protocol of superclasses without breaking into their secrets.

The connection primitives require the net from which the input is received and a connection specification (cf. above) defining defaults and details of interconnection. Fig. 11 depicts our interpretation of *bus_connect* and *x_connect* (left and middle).

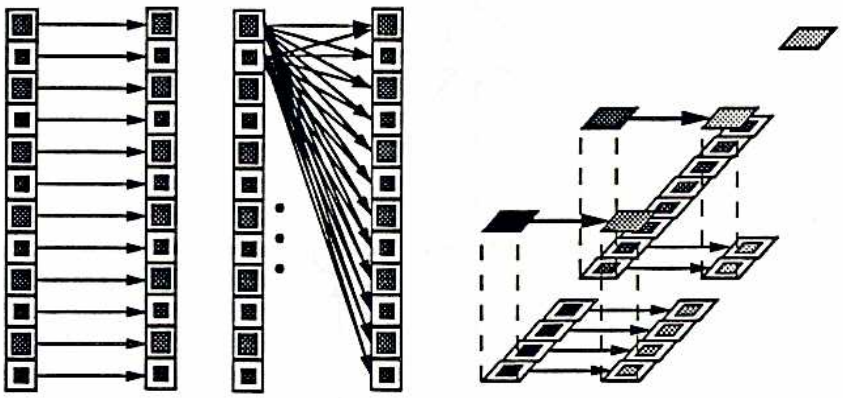

Figure 11: Connection routines

The drawing on the right illustrates how *bus_connect* translates over the levels of hierarchically composed nets. Implicitly the corresponding subobjects are aligned such that pairs of component sequences can be recursively translated into sequences

of pairs (connections). *complete_connect* mutually interconnects the components of a net. This is particularly used for mutual inhibitions.

The simulation of nets may depend on various variables that are part of the simulation profile: the number of steps to take in a run; the current temperature for nets whose computation or learning is temperature dependent and so on. The meaning of a single *step* on the net level depends on the net class. In general it involves all components, while a *micro_step* selects a single component. *Shake* changes the state of the net by some random deviation from the current state. The change depends on the model temperature variable. *To_fixpoint* makes the net run until it converges.

*Self_adapt* optionally determines the error of the output side and passes on the request to components which adapt their weights to the given error and in this course feedback errors to their sources of inputs. The order of the component adaptation depends on the learning rule of the class.

*Describe* and *View* support the user interface.

## 3.2 Units

The unit library contains a number of prefabricated components that can be synthesized simply by combining them through multiple inheritance.

Fig. 3 (left top) illustrates this synthetic view in which units are made up of three facets: state (shaded bubble, left top), function (top right) and interconnection (bottom). Choosing a specific class from the respective libraries allows us to form various combinations freely. A fair number of such combinations are prefabricated to offer standard off-the-shelf unit classes (cf. Fig. 12).

Nevertheless, according to our experience, users often require specific unit functions tuned to the problem at hand. A function package contains a number of basic unit functions for this purpose. The unit functionality includes

1. creation: *create_site* (cf. sites below);
2. state: *potential*, *output*;
3. connection: *connect*, *connectedp*, *disconnect*;
4. computation: *step*, *compute*, *post*, *reset*, *random_reset*;
5. learning: *self_adapt*, *clamp*, *release_clamp*;
6. visualization: *describe*, *view*.

*Connect* adds a single connection and *connectedp* is the corresponding predicate determining whether a pair of units is connected.

For the purpose of simultaneous updates (synchronous and asynchronous), a unit *step* is separated into the two phases of *compute* and *post* for all descendents of PHASE2_UNIT. In a step of simultaneous updates, all included units compute and

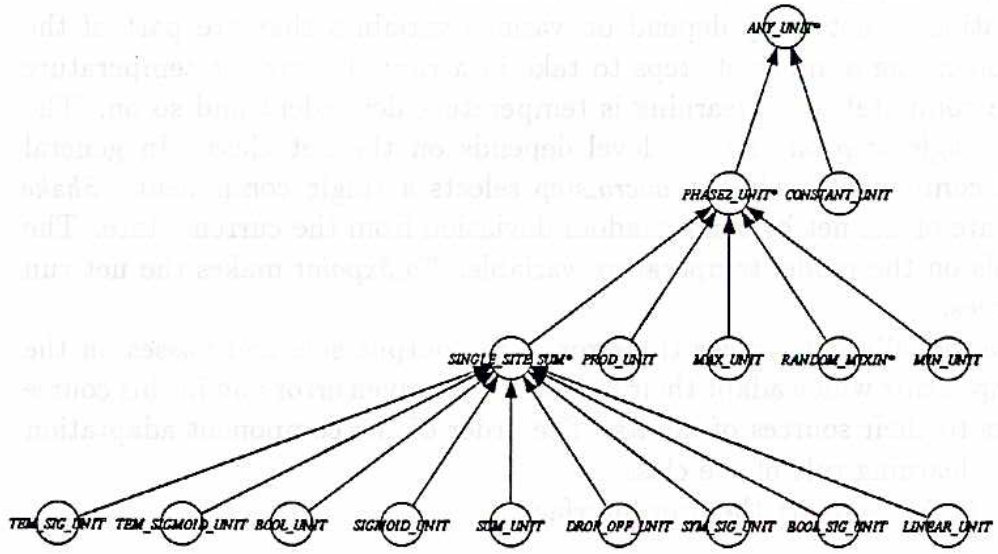


Figure 12: Part of the unit hierarchy.

then all included units post. For an interleaving semantics of asynchronous behavior this distinction is irrelevant since we assume that temporal measurements can be made infinitely fine and hence steps can always be ordered.

Clamping of units, i.e. freezing its state is supported by *clamp* and *release_clamp*. This can be used by some learning algorithms to force other units to internally adapt to a set of training patterns in order to distribute the internal representation of knowledge.

## 3.3 Sites

Sites represent the state of training which undergoes change only over a longer period of computation depending on the learning method supported by a net. From the structural viewpoint of composition, however, sites have a single output (inherit from class OUTPUT_OBJECT) like units.

Site operations mainly deal with connection. The major routines have the obvious interpretation:

1. creation: *create_site* (some classes only)
2. connection: *connect*, *disconnect*, *connectedp*,
   *site_connect*, *site_disconnect*, *site_connectedp*,
3. computation: *output*, *ith_input*;
4. training: *site_adapt*;

5. visualization: *describe*, *view*.

The routines with the prefix '*site*' only operate locally on the site; the unprefixed versions are interpreted as operations including the subsites in a site tree, if the current site is the root of some site tree. For instance, *disconnect(u)* would disconnect the input unit u from all terminals of a site tree, while *site_disconnect(u)* would erase only those connections terminating in the current site.

Sites can be weighted and unweighted. There are classes to support *sharing* of connections and/or of weights. Connection sharing is mainly a matter of compacting and centralizing the data representation of models. In contrast to this, weight sharing implies a different semantics of learning since multiple units interfere with each other by adapting the same weights. Fig. 13 shows part of the inheritance graph in the site library. The representative class SITE (right middle) supports the weighted sum of its inputs. All sites can receive input from any OUTPUT_OBJECT. This allows

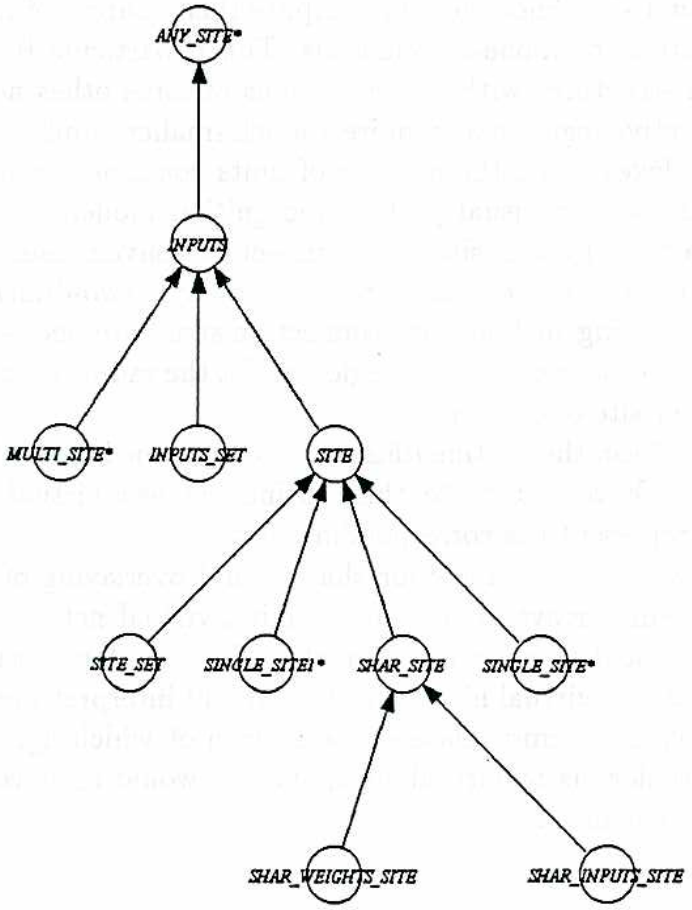


Figure 13: Part of Site Hierarchy

the user to compose complex interconnection patterns similar to dendritic trees with varying site functions at the accumulation points (cf. Fig. 3). *self_adapt* and routines of a weight (vector) object allow to modify weights.

Some classes at the leaves of the inheritance graph, like MULTI_SITE for example, represent a specific composite site structure that is automatically built when the object is created. MULTI_SITE, for instance, includes a vector of sites whose outputs it sums up. As part of its creation protocol, a MULTI_SITE creates and initializes a number of sites by iteratively calling back *create_site* which can be specialized by descendents.

## 3.4 Virtualization

All of the above class families support *virtualization*. Virtual objects 'simulate' the protocol of their class family. For instance, a virtual unit may behave like a unit in sending and receiving signals to and from other units and it may observe the connection protocol. But its state and inputs is not represented in the instance proper. The virtual unit will know to find the information in some shared and compact objects.

Similarly, some virtual sites know how to compute their sources of input rather than storing the connections on a one-by-one basis. This is particularly efficient for regular interconnection structures with convex regions of some other net. The determination of the respective region may require a much smaller number of reference points (units or unit indexes) than the number of units contained in it. Consider for instance an artificial net for visual pattern recognition modeling parts of the postretinal neural behavior. Typical sites will connect to convex, usually circular, regions of some plane of units. If the plane is represented by a two-dimensional matrix, a virtual site representing such an interconnection structure needs a reference to this matrix, an index to the center of the region and a the radius to calculate the input or supporting other site operations.

As part of the site protocol the routine *ith_source*[4] returns the ith unit from which a site receives its inputs. For a virtual site, this routine creates a virtual unit on the fly which 'pretends' to represent the corresponding unit.

Last but not least, virtual nets allow for sharing and overlaying of collections of units. Rather than using arrays of subnets or units, virtual nets refer to other nets and know how to calculate their components. Consider for instance a two-dimensional plane of units. A virtual hierarchical net might interpret square regions of 100-by-100 units, say, as its immediate subnets, each of which again falls into 10-by-10 subregions. Analogous to virtual sites, the net would retrieve the corresponding units by index calculation.

[4] in support of the tour view and also for feeding back error signals in a representation independent manner.

# 4 Examples

## 4.1 Layered Xor

The following example is a variation of the class XOR_NET presenting in a previous section. Here we use a layered net and show how higher level structure also lead to an abstraction with respect to interconnection. The input layer (XOR_INPUT) is a net consisting of the bottom two units. These are created following the same superclass-initialize-and-callback pattern that we used above. This is the only type-specific behavior of the class. The other two layers are the units proper[5].

```
class XOR_INPUT export repeat NET
inherit NET [BOOL_RND_UNIT] redefine step, create_unit

feature

    create is do net_init (2) end ;

    create_unit (i: INTEGER): bool_rnd_unit is do Result.create end ;

    step is do serial_step end ;

end
```

Here the class XOR_NET is modified to use the superclass LAYERED_NET which automatically creates the interconnection from layer to layer upon creation based on the connection specifications defined by the function *layer_connect_spec*. Only the connections from input to output layer remain to be wired in this variant of the model.

```
class XORNET export repeat LAYERED_NET
inherit LAYERED_NET [COMP_OBJECT]
        redefine step, create_component, layer_connect_spec

feature

    create is
      do
        net1d_init (3); - - create and x_connect 3 layers
        layer(0).connect_to (layer(2),con1);
      end ;
```

[5]Nets or units can be layers.

```
layer_connect_spec (i: INTEGER): ANY_CON is
  do
    inspect i - - name of connect target
    when 1 then Result:=con1
    when 2 then Result:=con2
    end ;
  end ;
...

end
```

## 4.2 Four colors: boolean view

This is a slight redefinition of the example developed in Section 2.3.4. Here we use boolean units (with the default boolean threshold of 0.5) which makes the net converge much faster. After all, according to our initial problem statement the units represent a binary relationship that either holds or does not hold. The activity based unit function in the previous example can be interpreted in a discrete way by distinguishing three ranges of unit activation in terms of the weighted sum $I$ of inputs:

1. $I \leq -5$, some other color of this region is selected.
2. $-5 < I < 0$, no other color in this region selected, but some neighbor has this color.
3. $I = 0$, no other color in this region selected and no competing neighbor.

We take advantage of this discrete interpretation in the redefinition of the predicate *deterministicp* that supports the simulator's reasoning about convergence of the computation.

```
class COL4UNIT inherit BOOL_UNIT redefine post, deterministicp

feature

  post is
    do
      if unif_rnd*strong_inhibition < potential then unit_flip_up
      else unit_flip_down end ;
    end ;
```

```
    deterministicp: BOOLEAN is
      do Result := ( potential <= strong_inhibition or potential = 0.0) end ;

end
```

In this solution to the problem, we redefine the *post* routine. The potential is the weighted sum of the inputs; *compute* is inherited from BOOL_UNIT. The potential and some non-determinism is used to determine the output. Also here we redefine *deterministicp*, a predicate that is called by the inherited routine *fixpointp*. *Fixpointp* is true only if a step will not change the unit's output *and* the unit is deterministic in the current input range.

## 4.3 Using multiple sites

The following example solves the four color problem with multi-site units. This variant uses two sites for the different kind of connections. The example also shows how to combine basic building blocks for units and demonstrates the default site connection protocol for multi-site units.

This time units inherit directly from ANY_UNIT. In class ANY_UNIT, *potential* and *output* are not separate attributes as in PHASE2_UNIT; rather *output* is a function of the potential. This is acceptable since in this model our computation mode of choice is asynchronous.

The two sites are built when the unit is created,

Moreover it is noteworthy that the weak inhibition connections need to be dynamically extended with each new neighbor relation being 'built' while the strong mutual inhibition within one region can be represented by a fixed size site – there are always four colors while there are a variable number of neighbors. In the connection protocol this is transparent.

```
class COL4UNIT inherit
    ANY_UNIT redefine post ...;
    MULTI_SITE [COL4UNIT] redefine create_site ...;
    COL4OBJ;
feature

    create is
      do
        multi_site_init (2); - - create two sites right now.
        reset;
      end ;
```

```
create_site (i: INTEGER): SITE [COL4UNIT] is
  local s: SITE [COL4UNIT];
  do s.create(4); Result:=s
  end ;

end
```

The other routines like *post* are identical to the previous example and are not shown here.

The selection of the proper site during connection building is achieved by a redefinition of the connection specifications of the class COL4OBJ, the class from which all classes in our example inherit. This is straightforward here since we used different connection specifications for the different strengths in our initial example.

```
strong_inhibit: CONST_CON is
  once
    Result.create (strong_inhibition,sys.always);
    Result.select_ith_site (0);
  end ;

weak_inhibit: const_con is
  once
    Result.create (weak_inhibition,sys.always);
    Result.select_ith_site (1);
  end ;
```

## 4.4 Using shared weights

In the four color example almost all weights are identical and can be shared. In the following version of the problem we exploit weight sharing. Due to the 'regularity' in the structure of our solution there are only two types of connections each with a type specific constant weight:

1. mutual inhibition of colors in the same region. The number of connections of this type per unit is fixed. The weight represents strong inhibition;

2. mutual weak inhibition of neighboring colors. The number of connections varies with the number of neighbors.

The example shows how sites with shared weights can be used to 'fold' all similar weight vectors into a single one. In terms of the connection protocol there is no difference between sharing and non-sharing sites. The type declaration for shared weight variable is different of course and the routine that creates the sites needs to be

redefined to create sites of this more specific class (SHAR_WEIGHTS_SITE). When a site is created, a reference to the shared weight vector is passed to *create*. The functions *color_inhibition* and *neighbor_inhibition* create or get the corresponding weight vectors from a class variable (Eiffel **once** ).

```
class SHAR_COL4UNIT inherit COL4UNIT redefine create_site

feature

    create_site (i: INTEGER): SITE [COL4UNIT] is
      local s: SHAR_WEIGHTS_SITE [COL4UNIT];
      do
        inspect i
        when 0 then s.create (color_inhibition);
        when 1 then s.create (neighbor_inhibition);
        end ;
        Result:=s;
      end ;

    color_inhibition: ARRAY_SET [REAL] is once Result.create (3) end ;

    neighbor_inhibition:ARRAY_SET [REAL] is once Result.create (4) end ;

end
```

# 5 Conclusion

We described the initial 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.

A number of concepts described and techniques chosen in the implementation are still subject of discussion. In particular we expect a change in the chosen implementation language from Eiffel to the Eiffel related Sather language developed at ICSI. Sather combines some selected features of Eiffel with a more efficient implementation and a less purist and restrictive approach towards object-oriented programming than Eiffel[11]. Another important argument: Sather will most likely be in the public domain, i.e. without severe copyright restrictions and licenses.

# Acknowledgement

I am grateful to Jerry Feldman with whom I had many stimulating discussions on the design. He also had worked out initial sketches of some classes when I came to ICSI. Thanks to Jeff Bilmes with whom I worked on the design of the user interface and who did some experiments with the Interviews library. I am also grateful to Susan Weber who worked with the simulator in the last three months. Without her, this report would not be what it is. She had many detailed ideas on how class interfaces could be improved. Also some examples are due to her.


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

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

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

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

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

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

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

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

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

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

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

[15] 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

[16] 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