THE CONNECT FRAMEWORK: A SIMULATION TOOL FOR NETWORKS OF COMMUNICATING OBJECTS

Gerd Kock
Fraunhofer Institute for Computer Architecture and Software Technology
Kekuléstr. 7, D 12489 Berlin, Germany
E-mail: gerd.kock@first.fraunhofer.de

KEYWORDS

Object Orientation, Modelling, Simulation, .NET

ABSTRACT

The CONNECT Framework presented in this paper is a simulation tool supporting the modelling of networks of communicating objects. It is implemented on the base of the .NET Technology. From the programmers point of view, it extends the C# language by a few attributes and offers a few classes. The paper introduces the new constructs and demonstrates their use by giving two examples. The first example is the Game of Life, and the second one, with special emphasis on signal based communication, is the Backpropagation Network.

INTRODUCTION

A modelling pattern found in areas like Artificial Neural Networks (Rojas 1996), Scientific Computing (Heath 2002), Swarm Intelligence (Kennedy and Eberhart 2001) and others requires working with arrays of communicating objects. Modelling large arrays of objects in a homogeneous manner is a non trivial problem. In some cases it is the index of an object that determines the processing (e.g. state-based manipulation or communication), in other cases all objects of an array are processed in a uniform way. A provision of special linguistic constructs for handling such arrays would allow for more elegant and error-proof modelling and programming of a whole range of different applications.

Modern software development usually takes place in some object oriented environment like the .NET Framework (Beer et al. 2003) or the Java 2 Enterprise Edition (Perrone et al. 2003). Considering the modelling pattern mentioned above in the context of such environments, we have to look at the concepts of classes and arrays, as any object is an instance of a class type, and as arrays are important means for establishing networks. The CONNECT Framework combines features from both of these concepts, and is an implementation for the .NET platform. The implementation uses the fact, that C# supports reflection, and that it is one of the first languages, which can be extended with the help of so called attributes (Drayton et al. 2003).

For any given type `T`, a corresponding *layer type* `Layer_T` can be generated, and these layer types reflect features of class and array types. To give an example consider class `Unit`:

```
[unit]
public class Unit {
  [item]
  public double myVar;
  [item]
  public void MyFun(double x) {...}
}
```

Attribute `[unit]` turns the class into a *unit type*, where attribute `[item]` turns the field `myVar` into an *item field* and the method `MyFun()` into an *item method*. The interface of the generated *layer type* `Layer_Unit` looks as follows:

```
public class Layer_Unit : Layer {
  // constructor
  public Layer_Unit(params int[] dim);
  // indexer
  public Unit this [params int[] ix]
    { get; set; }
  // lifted field
  public Layer_Double myVar { get; }
  // lifted method
  public void MyFun(System.Double x);
}
```

With the aid of the constructor and the indexer layer objects can be instantiated and used similar to array objects. For example, let us create a two dimensional *base layer*:

```
Layer_Unit ulayer;
ulayer = new Layer_Unit(m,n);
```

The item field `myVar` (or item method `myFun()`) with index `[i,j]` can be accessed as follows:

```
ulayer[i,j].myVar = 3.14;
ulayer[i,j].MyFun(3.14);
```

Proceedings 18th European Simulation Multiconference
Graham Horton (c) SCS Europe, 2004
ISBN 3-936150-35-4 (book) / ISBN 3-936150-36-2 (CD)

```
[unit] public class Life {
   [item] public int state;
   [item] public Life[] vector;

   [item] public void InitState(Random r) { state = r.Next(2); }

   private int next;
   [item] public void NextState()
      int sum = 0;
      for (int i = 0; i < vector.Length; i++) { sum += vector[i].state; }
      if (sum < 2 || sum > 3) { next = 0; }
         else if (sum == 3)   { next = 1; }
         else                 { next = state; }
   }

   [item] public void UpdateState() { state = next; }
}
```

Figure 1: The unit type Life

But layer types are not identical to array types. If they are generated from unit types, the public fields and methods being marked by attribute `[item]` are "lifted" to the interface of the layer type. If, for a given layer instance, the lifted identifiers are used to access the layer, actually all corresponding fields or methods of the layer elements are addressed. In the case of methods, this is a kind of shorthand for a loop. Invoking a lifted method results in invoking the underlying unit method for all elements of the given layer:

```
ulayer.MyFun(3.14);
```

A lifted field is an *item layer* (e.g. `ulayer.myVar` has type `Layer_Double`), which opens up the possibility to invoke methods defined for layer types. For example, the overloaded method `Set()`, which can be used to initialize all layer elements with the same constant:

```
ulayer.myVar.Set(3.14);
```

Accessing lifted items can be combined neatly with the fact, that the underlying unit items can also be accessed by index. This is used by the CONNECT Framework to offer powerful means for connecting (or setting) the item fields of layers. The parameters of the corresponding methods are, on the one hand, lifted item fields, and on the other hand, relations or functions over indices.

The next section presents the CONNECT Framework components. After that, it is explained in detail how layer items can be connected, and the Game of Life is used to demonstrate this process. Then the focus is switched to signal based communication, Here, as an example, the Backpropagation Network is given. The concluding section summarizes the lessons learned from the two examples and relates the CONNECT Framework to previous work of the author. At the end, a description of further research can be found.

CONNECT FRAMEWORK COMPONENTS

The CONNECT Framework consists of a library and a command line tool. Using the framework for the modelling of an application can be sketched as follows.

At first, the unit types of the application have to be identified. Let file `app.units` contain the source code of these types (class `Unit`, ...). Except for the attributes `[unit]` and `[item]` this is usual C# code.

Then, the command line tool can be used to generate all implied layer types (`Layer_Unit`, `Layer_Double`, ...), and to put them into file `app.layers`. The generation process relies on the types provided by the CONNECT library.

The rest of the application's source code can use the unit types as well as the generated layer types.

The CONNECT library includes class `Layer` (the base class of all layer types), signal classes for simple types like `int`, `double` etc., and class `Generator`. Signal classes are explained below. Class `Generator` contains methods for generating layer or signal types. These methods are provided for the rare cases, where the command line tool or the predefined classes do not suffice.

CONNECTING LAYER ITEMS

For discussing the way, in which objects can be connected, let us start with unit type `Elem`:

```
[unit]
public class Elem {
```

```
public class LifeNetwork {
   private Layer_Life layer;

   public LifeNetwork (int n, int m) {
      layer = new Layer_Life(n, m);
      layer.vector.Connect(layer, new Relation(IsNeighbour));
   }

   public void InitPopulation() {
      Random r = new Random();
      layer.InitState(r);
   }

   public void NextGeneration() {
      layer.NextState();
      layer.UpdateState();
   }

   private bool IsNeighbour(int[] x, int[] y) {
      if ( (x[0]!=y[0] || x[1]!=y[1]) && ( Math.Abs(x[0]-y[0])<=1 ) && ( Math.Abs(x[1]-y[1])<=1 ) )
         return true;
      else
         return false;
   }
}
```

Figure 2: Using the layer type Layer_Life

```
   [item] public Elem   transpose;
   [item] public Elem[] line;
}
```

Our goal is to organize `Elem` objects into a square matrix,

```
Layer_Elem matrix = new Layer_Elem(n,n);
```

such that the item layers `matrix.transpose` and `matrix.line` correspond to the transpose or lines, respectively, of that matrix. For a given line `i` and for all `j` and `k`, the one dimensional arrays `matrix[i,j].line` and `matrix[i,k].line` would contain the same objects.

To establish the requested connections, a function and a relation over indices are needed:

```
int[] transpose( int[] ix ) {
  int[] val = { ix[1], ix[0] };
  return val;
}

bool line( int[] ix, int[] iy ) {
  if (ix[0]==iy[0]) return true;
  return false;
}
```

The CONNECT Framework includes special types for such functions and relations:

```
Function fun = new Function(transpose);
Relation rel = new Relation(line);
```

Now, the layer methods `Set()` and `Connect()` can be applied:

```
matrix.transpose.Set(matrix, fun);
matrix.line.Connect(matrix, rel);
```

To explain the meaning of `Set()`, let `ix` be an index, for which `transpose(ix)` is defined. Then `matrix[ix].transpose` is set to `matrix[transpose(ix)]`.

To explain the meaning of `Connect()`, let `iy` be another index, and let the result of `line(ix,iy)` be `true`. Then `matrix[ix].line` is extended by one component, and this component is set to `matrix[iy]`.

THE GAME OF LIFE

The first example is the famous Game of Life (Gardner 1970). The complete modelling can be found in Figures 1 and 2. Figure 1 shows the definition of unit type `Life`, and Figure 2 shows, how the generated type `Layer_Life` can be used to define a class `LifeNetwork`.

A unit of type `Life` has two public item fields and three public item methods.

Item `state` is used to signal, whether a unit is alive (`state==1`) or dead (`state==0`), and the one dimensional array `vector` is used to get the corresponding states of the immediate neighbours — where item method `NextState()` assumes, that the `vector` has been set accordingly.

Method `InitState()` is used to randomly set the state, method `NextState()` checks, how many of the neighbours are alive, and uses the result to set the private variable `next`, and method `UpdateState()` updates the state.

In class `LifeNetwork`, a private layer of type `Layer_Life` can be found. The constructor creates a two dimensional layer, and establishes the necessary connections. In this, the relation `IsNeighbour()` is used. For indices `ix` and `iy` the result `IsNeighbour(ix,iy)` is `true` exactly in the case, that `ix` and `iy` are the indices of immediate neighbours.

The meaning of the methods of type `LifeNetwork` is straightforward: `InitPopulation()` initializes the network, and `NextGeneration()` computes the next generation.

SIGNAL BASED COMMUNICATION

In the example given above, a `Life` unit communicates by reading the `state` of neighbouring units with the help of item field `vector` (see Figure 1). In a sense, the field `vector` allows a `Life` unit to access the "outside world".

However, it might be advantageous to communicate just by sending and receiving signals, without accessing the state of other units. To support this procedure, the CONNECT Framework offers the possibility to generate signal classes. More precisely, for any given type `T`, classes `Fanout_T`, `IVector_T`, and `OVector_T` can be generated. (For simple types like `double`, the signal classes are part of the CONNECT library.)

The signal classes represent *fanout signals*, *input* and *output vectors*. To give an example, let us start with three unit types based on the signal classes generated from type `double`:

```
[unit] public class FUnit {
  [item] public Fanout_Double y;
}

[unit] public class IUnit {
  [item] public IVector_Double ivec;
}

[unit] public class OUnit {
  [item] public OVector_Double ovec;
}
```

We consider the layers `flayer` (of type `Layer_FUnit`), `ilayer` (type `Layer_IUnit`) and `olayer` (type `Layer_OUnit`). At first, `n` fanout signals are created and initialized:

```
flayer = new Layer_FUnit(n);
for (int i = 0; i < n; i++)
   flayer[i].y.Value = i;
```

Then, `n` input and output vectors are created:

```
ilayer = new Layer_IUnit(n);
olayer = new Layer_OUnit(n);
```

Initially, these vectors are empty. However, input vectors can be connected with fanout signals and output vectors analogous to the way described above. Let `Full` be the relation, which includes all index pairs, i.e. for all pairs `(ix,iy)` we have:

```
Full(ix, iy) == true
```

This relation is used for establishing connections:

```
Relation full = new Relation(Full);
ilayer.ivec.Connect(flayer.y, full);
```

As all index pairs belong to the relation `Full`, the result of this connect statement is, that for all `0 <= i,j < n` the input vector `ilayer[i].ivec` is connected with fanout signal `flayer[j].y`. This means, the input vector is extended by one component, and this component is set to the fanout signal. After that, all input vectors have `n` components and, as the connections are established in ascending order, we have:

```
ilayer[i].ivec[j] == j
```

Modifying the fanout signals

```
for (int i = 0; i < n; i++)
   flayer[i].y.Value = -i;
```

has an immediate consequence:

```
ilayer[i].ivec[j] == -j
```

Now, the input vectors are connected to the output vectors:

```
ilayer.ivec.Connect(olayer.ovec,full);
```

Again, the connections are established in ascending order. For all `0 <= i,j < n` the input vector `ilayer[i].ivec` is connected with output vector `olayer[j].ovec`. Here, both vectors are extended by one component, and the new component of input vector `ilayer[i].ivec` refers to the new component of output vector `olayer[j].ovec`. This results in `n+n` components

for each input, and `n` components for each output vector.

By default, the new components of the output vectors are set to zero. So we have:

```
ilayer[i].ivec[n+j] == 0
```

If we assign other values to the output vectors,

```
olayer[i].ovec[j] = i*i;
```

we have:

```
ilayer[i].ivec[n+j] == j*j
```

THE BACKPROPAGATION NETWORK

The Backpropagation (BP) Network is capable of learning a functional mapping $x \mapsto y$. In this, network input $x = (x_1, \ldots, x_i)$ and network output $y = (y_1, \ldots, y_o)$ are vectors of numbers. The BP Network can be applied in areas such as sensor processing, pattern recognition, data analysis, and control (Rojas 1996).

Such a network consists of one input, one or more hidden, and one output layer. In the forward pass, the input layer is used to present the network input x to hidden layer 1. For $n \geq 1$, hidden layer n does some processing and presents its results to hidden layer $n+1$ or, finally, to the output layer. The output layer does an analogous processing to compute the network output y.

The output of a network depends on the weights, where each hidden and output unit is associated with one bias weight and one weight for each incoming signal.

For a given set of training data $\{(x, t)\}$ such a network is able to learn the mapping $x \mapsto t$ where $t = (t_1, \ldots, t_o)$ is a target vector. Learning means to adapt the weights correspondingly. In general, learning a good mapping of the given targets is not the only goal of training. Another goal is that the network is able to "generalize", i.e. after training it should be able to map an input x, which has not yet been seen so far, in a "sensible" way to an output y.

During training, the result $y = (y_1, \ldots, y_o)$ of mapping a sample $x = (x_1, \ldots, x_i)$ is compared with the associated target $t = (t_1, \ldots, t_o)$. The backpropagation algorithm essentially is a gradient descent method minimizing the quadratic error measure $\sum_{i=1}^{o}(t_i - y_i)^2$ (seen as a function of the weights). Learning can be done either in online or in batch mode. Within online learning, the weights are adjusted each time, a training example has been presented; within batch learning, weight adjustment takes place only after all training examples have been seen. Once a network is trained, a recall is done in one forward pass.

Now we consider, how the CONNECT Framework can be used to program a Backpropagation Network (implementing online learning). In Figure 3 the unit types `Input`, `Hidden`, and `Output` can be found, and Figure 4 contains the class `BPNetwork`, which is based on these types.

The only item of type `Input` is a fanout signal `y`, which is used to present the network input to hidden layer 1. The types `Hidden` and `Output` both are derived from class `BPUnit`, which contains all common elements:

- a fanout signal `y` used to present the result to the next layer or as network output, respectively;
- an input vector `x` for collecting the fanout signals of the preceding layer, and an associated bias `b` and weight vector `w`;
- an output vector `outErr` used to send back error signals.

All layers (except for the input layer and hidden layer 1) send back error signals to the preceding layer. It is important, that error signal `outErr[i]` is send to the unit, where input `x[i]` stems from. Specifically, the size of error vector `outErr` either is 0 (for hidden layer 1) or has to coincide with the sizes of the vectors `x` and `w`.

The item methods `AdjustVector()` and `InitWeights()` are used during network initialization. For each incoming signal, a weight component is established, and all weights are initialized with random numbers in between 0 and 1.

Item method `Forward()` performs the computation, which is done in the forward pass. Actually, the so called net input `net = w * x + b` is mapped by the sigmoid function. In this, `w * x` is the vector dot product and the result of `Sigmoid(net)` is `1 / (1 + exp(-net))`.

Item method `Backward()` is used for learning. The parameters are the so called learning rate `eta` and a value `delta`, which depends on the given target and is computed differently for hidden and output layers. Based on the given parameters, at first the error signals for the previous layer are set, and then the weights are modified.

```
[unit] public class Input {
   [item] public Fanout_Double y = new Fanout_Double();
}

public class BPUnit {
   [item] public Fanout_Double y = new Fanout_Double();

   [item] public IVector_Double x = new IVector_Double();
   [item] public double   b;
   [item] public double[] w;
   [item] public OVector_Double outErr = new OVector_Double();

   [item] public void AdjustWeightVector() { w = new double[x.Length]; }
   [item] public void InitWeights( Random r ) {
      b = r.NextDouble();
      for (int i = 0; i < x.Length; i++ ) w[i] = r.NextDouble();
   }

   [item] public void Forward() { y.Value = NN.Sigmoid( w * x + b ); }

   public void Backward(double eta, double delta) {
      for (int i = 0; i < outErr.Length; i++ ) outErr[i] = delta * w[i];
      b = b + eta * delta;
      for (int i = 0; i < x.Length; i++ ) w[i] = w[i] + eta * delta * x[i];
   }
}

[unit] public class Hidden : BPUnit {
   [item] public IVector_Double inErr = new IVector_Double();
   [item] public void Backward(double eta) {
      double delta = y * (1-y) * inErr.Sum;
      base.Backward(eta, delta);
   }
}

[unit] public class Output : BPUnit {
   [item] public double t;
   [item] public void Backward(double eta) {
      double delta = y * (1-y) * (t-y);
      base.Backward(eta, delta);
   }
}
```

Figure 3: The unit types for the Backpropagation Network

In the derived types `Hidden` and `Output` the value `delta` is computed and passed to the `Backward()` method of type `BPUnit`. In one case (type `Output`) the value depends on the difference between target and network output (`t-y`), and in the other case (type `Hidden`) it depends on the sum of incoming error signals (`inErr.Sum`).

In Figure 4, a Backpropagation Network with two hidden layer is presented. At first, we find the private layer fields `input`, `hidden1`, `hidden2`, and `output`. The constructor creates a network with `i` input and `o` output units, establishes the necessary connections between the layers, and performs the corresponding adaptations of the weight vectors.

Method `InitWeights()` initializes the weights randomly, method `Forward()` implements the forward and method `Backward()` implements the backward pass. Note the details of the `Forward()` and `Backward()` methods.

The parameter `x` of network method `Forward()` is the network input, which is assigned to the fanout layer `input.y`. Then, method `Forward()` is called for hidden layers 1 and 2, and finally for the output layer. At the end, the fanout layer `output.y` is returned as network result.

The parameters `eta` and `t` of network method `Backward()` are the learning rate or target vector, respectively. The target vector is assigned to the item layer `output.t`. Then, method `Backward()` is called for the output layer, and for hidden layers 2 and 1. At the end, function `sqdist()` computes the quadratic error, which is returned as the result.

```
public class BPNetwork {
   private Layer_Input input;
   private Layer_Hidden hidden1, hidden2;
   private Layer_Output output;

   public BPNetwork( int i, int h1, int h2, int o ) {
      input   = new Layer_Input(i);
      hidden1 = new Layer_Hidden(h1);
      hidden2 = new Layer_Hidden(h2);
      output  = new Layer_Output(o);

      hidden1.x.Connect( input.y, new Relation(Full) );
      hidden1.inErr.Connect( hidden2.outErr, new Relation(Full) );
      hidden2.x.Connect( hidden1.y, new Relation(Full) );
      hidden2.inErr.Connect( output.outErr, new Relation(Full) );
      output.x.Connect( hidden2.y, new Relation(Full) );

      hidden1.AdjustWeightVector();
      hidden2.AdjustWeightVector();
      output.AdjustWeightVector();
   }
   public void InitWeights() {
      Random r = new Random(1);
      hidden1.InitWeights(r); hidden2.InitWeights(r); output.InitWeights(r);
   }
   public double[] Forward( double[] x ) {
      double[] y = new double[output.Length];
      for (int i = 0; i < input.Length; i++ ) input.y[i].Value = x[i];
      hidden1.Forward(); hidden2.Forward(); output.Forward();
      for (int i = 0; i < output.Length; i++ ) y[i] = output.y[i];
      return y;
   }
   public double Backward( double eta, double[] t ) {
      for (int i = 0; i < output.Length; i++ ) output.t[i] = t[i];
      output.Backward(eta); hidden2.Backward(eta); hidden1.Backward(eta);
      return sqdist( output.t, output.y );
   }
}
```

Figure 4: The Backpropagation Network

CONCLUSIONS

The Game of Life (Gardner 1970) and the Backpropagation Network (Rojas 1996) have been used to demonstrate how the CONNECT Framework can be employed for modelling applications. The examples show that this can be done in a compact and elegant way, such that the resulting C# programs almost have the character of specifications.

Modelling a network of communicating objects can be done by considering a given application from two points of view. On the one hand, one can consider the functionality of the objects of that application, and in this can take the necessary communication structure as given. And on the other hand, one can take the functionality of the objects as given, and can concentrate on the global aspects of connecting objects and invoking complete layers of them.

The predecessor of the CONNECT Framework is the Neural Network (NN) description language CONNECT, which had been developed in the nineties. A compiler translating CONNECT specifications into C++ classes is the software kernel of the NeuroLution system, which integrates hardware and software components (Kock et al. 1999).

The NN language CONNECT allows for flexible definitions of networks of simple processing units, each of them communicating with the others by sending simple signals, and can be applied to design neural networks, cellular automata as well as other simple distributed systems (Fabiunke and Kock 1999). But communication is restricted to simple signal types, and it is not possible to connect (or set) other than signal items.

The NN language CONNECT is a specific domain language, where the CONNECT Framework actually is a simple extension of the C# language by the two attributes `[unit]` and `[item]`. Using these attributes allows for solutions, the abstraction level of which is high and makes it simple to

reuse them. As the modelling language is C#, there is no restriction on the domain of possible applications.

FURTHER RESEARCH

The two examples given in this paper stem from the area of swarm intelligence (Kennedy and Eberhart 2001) or Artificial Neural Networks (Rojas 1996), respectively. For the latter domain it already has been proven, that many network models can be programmed easily by using the CONNECT Framework. The corresponding specifications written in the Neural Network description language CONNECT (Kock et al. 1999) can be translated easily into corresponding C# programs — which use the CONNECT Framework. Other possible application domains are Scientific Computing (Heath 2002) and Web Services (Alonso et al. 2004).

All domains mentioned above will be studied, and key applications will be developed to trigger further implementation and theoretical issues.

There are several implementation issues. One goal is to bring the current implementation into a product version state. Also, the use of threads and a distributed implementation will be considered. With respect to the generics of C# 2.0, a future version of the CONNECT Framework will include parameterized types `Layer<Type>`, `Fanout<Type>`, `IVector<Type>` and `OVector<Type>` — as an alternative for the `Generator` class mentioned above. Finally, an implementation for the Java platform will be considered.

Two kinds of "theoretical" issues occur. At first, the "final" form of the interface has to be fixed; this refers to the work out of layer methods for accessing and connecting layers, or to the question, whether beside `Connect()` methods there also should be `Disconnect()` methods, etc. Secondly, the meaning of networks consisting of interconnected layers mainly depends on the meaning of unit and layer types and on the meaning of layer methods, which globally access and (dis-)connect layer items; it might be helpful to develop some mathematical means for treating the formal semantics of such networks.

AUTHOR BIOGRAPHY

GERD KOCK studied Mathematics and Economy (University of Münster). For a few years, he worked for Siemens in Munich. His doctoral thesis was in Computer Science (Technical University of Karlruhe). Since 1992, he acts as senior researcher at the Fraunhofer Institute for Computer Architecture and Software Technology (Fraunhofer FIRST) in Berlin. For many years, he worked in the areas of Programming Languages, Compilers and Software Technology. In cooperation with industry, he developed software concepts for a Neural Network simulation tool. He gave lectures and seminars at many places, e.g. at the Humboldt University of Berlin and at the Technical University of Berlin.

REFERENCES

Alonso, G.; F. Casati; H. Kuno; and V. Machiraju. 2004. *Web Services. Concepts, Architectures and Applications.* Springer.

Beer, W.; D. Birngruber; H. Mössenböck; and A. Wöß. 2003. *Die .NET-Technologie.* dpunkt.verlag.

Bonabeau, E. and C. Meyer. 2001. "Swarm Intelligence: A Whole New Way to Think About Business". *Harvard Business Review (May)*, 107–114.

Drayton, P.; B. Albahari; and T. Neward. 2003. *C# in a Nutshell.* O'Reilly.

Fabiunke, M. and G. Kock. 1999. "A Connectionist Method to Solve Job Shop Problems". *Second International Conference on Intelligente Processing and Manufacturing of Materials* (IPMM'99), Big Island, Hawaii, July 1999.

Gardner, M. 1970. "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"". *Scientific American*, 120–123.

Heath, M.T. 2002. *Scientific Computing: An Introductory Survey.* McGraw-Hill.

Kennedy, J. and R.C. Eberhart. 2001. *Swarm Intelligence.* Morgan Kaufmann Publishers.

Kock, G.; T. Fischer; W. Eppler; H. Gemmeke; and T. Becher. 1999. "NeuroLution: Integrated Hardware and Software for the Development of Neural Network Applications". *Systems Analysis – Modelling – Simulation* 35, No. 4, 447–481. Gordon And Breach Science Publishers.

Perrone, P.; S.R. Venkata; and T. Schwenk. 2003. *J2EE Developers's Handbook.* SAMS.

Rojas, R. 1996, *Neural Networks - A Systematic Introduction.* Springer.

Winder, R. and G. Roberts. 2000. *Developing Java Software, 2nd Edition.* Wiley.