Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep Informatica

Creating and manipulating hierarchically structured neural networks

Eltjo Voorhoeve

advisors:
Dr.ir. J.A.G. Nijhuis
Ir. W.J. Jansen
Drs. J.H.M. Seinhorst

June 1997

RuG

Voor mamma

Samenvatting

Neurale netwerken worden steeds belangrijker voor het oplossen van non–lineaire problemen. Twee principes worden al gebruikt voor het oplossen van problemen met neurale netwerken: oplossen met behulp van voorkennis en het "divide and conquer" principe. Met behulp van voorkennis is het mogelijk een gunstige beginsituatie van een neuraal netwerk te bepalen. Het "divide and conquer" principe maakt het mogelijk een groot probleem te verdelen in meerdere kleine problemen die apart opgelost kunnen worden. Het samenvoegen van deze kleinere problemen resulteert dan weer in de totale oplossing van het probleem. Beide principes worden al toegepast, maar de combinatie van beiden is nog een vrij nieuw onderzoeksgebied. Het gebruik van deze principes vraagt om een netwerkstructuur die hiërarchisch is opgebouwd.

Het simuleren van neurale netwerken gebeurt meestal met behulp van éen simulatieprogramma. Binnen de vakgroep informatica aan de Rijksuniversiteit Groningen wordt vrij veel gebruik gemaakt van het programma InterAct. Dit programma wordt ook voor een groot deel daar ontwikkeld.

Het doel van de afstudeeropdracht is nu om binnen InterAct een grafische userinterface te ontwikkelen die het mogelijk maakt om neurale netwerken met een hiërarchische structuur te creëren en manipuleren.

Abstract

Neural networks have become more and more important for solving non–linear problems. When solving a problem with neural networks, two methods are already used: prior knowlegde and the “divide and conquer” algorithm. Solving a problem using prior knowlegde makes it possible to determine a suitable initialization of a neural network. The “divide and conquer” method makes it possible to divide a big problem into several smaller problems, which can be solved separately. Merging the solutions of these smaller problems results in the total solution of the problem. Both methods are already used, but the combination of both is quite new area of research. Using these methods demands a network that is hierarchically structured.

Simulating neural networks is often done by using an existing simulator. At the department of computing science at the Rijksuniversiteit Groningen, the simulator InterAct is often used. This program is also being developed there.

The goal of this thesis is to create a graphical user interface within InterAct to make it possible to create and manipulate neural networks that are hierarchically structured.

Preface

During my study computing science at the state university of Groningen, my interest for both artificial neural networks and graphical user interfaces has been increased. After finishing my last courses, Dr.ir. J.A.G. Nijhuis offered me the possibility to develop a graphical user interface for InterAct, a simulator for neural networks. This program is both used and developed at the department of computing science at the state university of Groningen. The interface should make it possible to create and manipulate hierarchically structured neural networks.

Making this thesis was not possible without the support of some persons, who I would like to thank here in this preface. Special thanks to Dr.ir. J.A.G. Nijhuis, Ir. W.J. Jansen and Drs. J.H.M. Seinhorst for the guidance during this project. Furthermore I would like to thank my father, Rutger Lubbers and Jan Willem Omlo for the feedback while writing this thesis.

Finally I would like to thank my parents for their support all these years and my friends for sending me away at coffee breaks.

Groningen, June 1997
Eltjo Voorhoeve.

Chapter 1 Introduction

The ability to learn complex non–linear relationships between a number of inputs and a number of outputs is the main reason why Artificial Neural Networks (ANNs) are a very powerful tool for solving problems in a wide area of applications. ANNs often offer better solutions to non–linear problems than classical, often mathematical methods. Research on neural networks has been motivated from the fact that the human brain computes in an entirely different way than digital computers. It can perform recognition tasks somewhere within 100–200 ms, whereas tasks with a much lesser complexity will take days on the fastest computer in existence today. A human brain consists of approximately 10 billion neurons. During life, the human brain developes interconnections (or *synapses*) between these neurons. The process of the forming and the adjustment of synapses is called *learning*.

In this thesis we will discuss the creation of a graphical user interface which makes it possible to interactively create and manipulate artificial neural networks that are built up hierarchically.

This chapter will give an overview of the goal of this thesis and its basis. First, artificial neural networks are explained. Section 1.2 introduces hierarchical artificial neural networks and explains the need for artificial neural networks that are built up hierarchically. Section 1.3 points out how artificial neural networks can be designed. Finally, section 1.4 gives a description of the goal of this thesis.

1.1 Artificial Neural Networks

There are several architectures for artificial neural networks (or briefly "neural networks"). To explain the working of a neural network, we will look at a specific architecture, called the Multi Layer Perceptron (MLP) neural network, which uses a learning algorithm called error Back Propagation (BP). This architecture is very popular and is used in many applications. The MLP neural networks consist of three layers, each layer composed of one or more neurons. The *input layer*, which receives the input patterns, is the interface with the outside world. This layer is connected to the *hidden layers*, which are connected to the *output layer*. Figure 1.1 shows the architecture of a Multi Layer Perceptron. Input neurons are visualized as triangles, hidden neurons as squares and output neurons as circles. In the remaining of this thesis this visualization will be used.

Each neuron in an MLP neural network receives one or more input signals from the incoming synapses. These input signals are summed, weighted by the synapses. Given these summed input

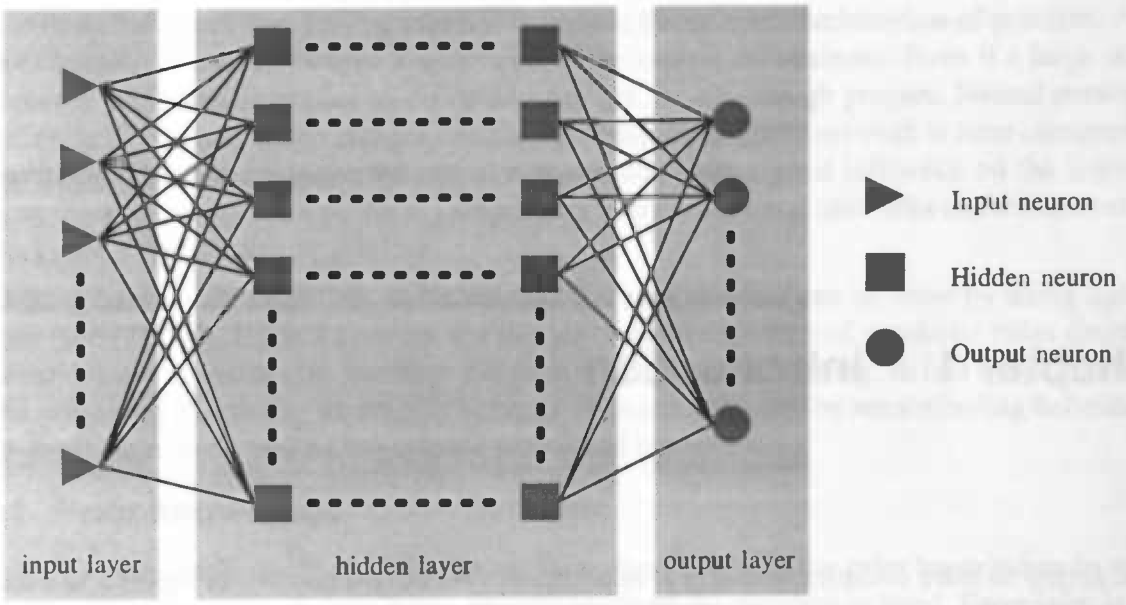


Figure 1.1: *Architecture of an MLP neural network*

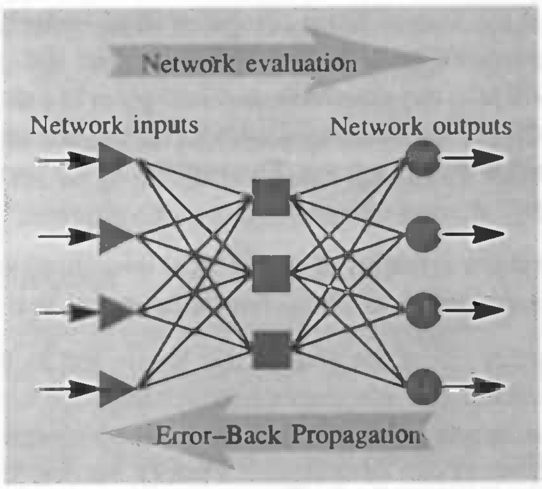


Figure 1.2: *Learning in an MLP neural network using error back propagation*

signals, the neuron makes calculations which are presented to the outgoing synapses, which are connected to the next layer. One calculation–cycle of a neural network consists of the calculations of all layers until the last calculations are presented to the output layer.

During the learning procedure of such an MLP network using error back propagation, the pattern generated at the output layer will be compared with the desired output pattern. After this, the internal weights of the synapses will be adjusted from the output layer to the input layer (this is the error *back* propagation algorithm), so when a similar pattern is presented to the input layer, the network will generate a "better" output pattern. This procedure is repeated until the difference between the output pattern and the desired target pattern is small enough (see Figure 1.2 for an MLP network with three layers). When offering an input pattern to the input neurons, the output neurons of a trained MLP neural network will provide the calculated output pattern. In the next

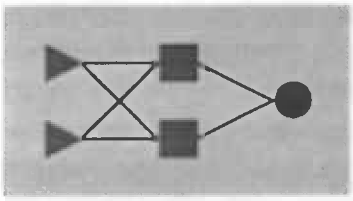


Figure 1.3: *MLP neural network solving the XOR–problem*

subsection we will investigate an example of an MLP neural network using BP. More information about neural networks can be found in [1].

1.1.1 A simple example

To show the working of a Multi Layer Perceptron neural network using error Back Propagation, we look at the XOR–problem. XOR stands for eXclusive–OR. It is an operation with two inputs, generating one output. The output can be 0 or 1, depending on the inputs. When the inputs are different, the XOR will give 1 as result. When the inputs are equal, the result will be 0. The next table shows the possible inputs and their outputs:

Input #1	Input #2	Output
0	0	0
0	1	1
1	0	1
1	1	0

This algorithm can be learned by a neural network. The MLP neural network that can deal with this algorithm is viewed in Figure 1.3. The algorithm to "learn" the XOR is:

1. Give a random input pattern to the input neurons (0–0, 0–1, 1–0 or 1–1);
2. Let each layer make their calculations;
3. Compare the output neurons with the desired output pattern;
4. If the difference between the output neurons and the desired output pattern is too big, use the BP–algorithm to adjust internal weights and return to step 1.

1.2 Design philosophies

Teaching a human to recognize a specific class of objects can be done in two different ways: by giving a *domain theory* which describes the characteristics of class members and their interaction or by giving lots of examples while telling whether each example is a member of the class or not. Also teaching a computer to get a problem–specific expertise can be done in these two ways. The terms associated with these two methods are *hand–built classifiers* (e.g. expert systems) and *empirical learning* (e.g. neural networks). Each of the two learning strategies have their disadvantages. Hand–built classifiers assume their domain theory is complete (which, in the real world, will be extremely difficult to achieve) and making a complete and correct classifier will probably take thousands of lines (possibly recursive). Predictions about the impact of changes in a single rule will almost be impossible. A more comprehensive discussion concerning these technique is found in [2].

Also empirical learning has its disadvantages. Examples to an empirical learning system are often offered by features (the characteristics of a system). An unbounded number of features can be used to describe a problem, but the user has to choose the correct combination of practice. Also the construction of useful features is a difficult, time consuming business. Even if a large set of examples is available, examples of exceptions are not, or not enough present. Neural networks in particular have some disadvantages themselves: training a neural network is time consuming, the parameters used to initialize the neural network can have a great influence on the learning process, there does not exist a problem independent design for neural networks and trained neural networks are very difficult to interpret.

Combining hand–built classifiers and empirical learning systems can be done by using *hybrid systems* (see [2], [3]). Hybrid systems use the set of hand constructed symbolic rules (used to make hand–built classifiers) to initialize a system with empirical learning. In fact hybrid systems use the same kind of learning strategy as humans. Humans also learn by remembering definitions (hand–built classifiers) and by experience (empirical learning).

1.2.1 Prior knowledge

Designing a domain theory to describe the problem, there is a need for prior knowledge. In most problem solving cases we already know something about the problem at hand. Especially in industrial applications there is mostly a human operator who can recognize special events and advise in the control of processes. This knowledge can be used as "prior knowledge" to solve the whole problem more easily.

An example of prior knowledge (also called expert knowledge) is an application which examines photographs of apples and has to recognize whether an apple is healthy or not. Prior knowledge can be used to interpret colors that can be expected on the photograph (e.g. background color, "proper apple colors" like red or green, brown from the stalk). Also information about the expected position and size of an apple can be used as prior knowledge.

1.2.2 Divide and conquer

Many complex problems are solved using the "divide and conquer"–method. Dividing a complex problem into several smaller problems will make the problem clearly structured. The smaller problems are easier to solve and their solutions can be combined to solve the whole problem.

We will now give an example of the "divide and conquer"–method. Suppose you've got photographs of cars of which you want to recognize the license plates. It is possible to divide this problem into two subproblems:

- Find a license plate within a photograph of a car;
- Recognize the letters and numbers on a license plate.

The newly created subproblems are less complex and thus easier to solve. When algorithms have been found for both problems, the output of the first algorithm can be used as an input for the second algorithm, which gives the solution of the whole problem.

Solutions obtained by dividing a problem into smaller subproblems will have a modular structure with a certain hierarchy. In the example above we have one algorithm solving the whole problem, but this algorithm is divided into two separate algorithms dealing with two subproblems.

1.2.3 Design approach

Both aspects described above, the use of prior knowledge (section 1.2.1) and modularity (section 1.2.2), are used separately in the design of neural networks, but the combination of both is quite

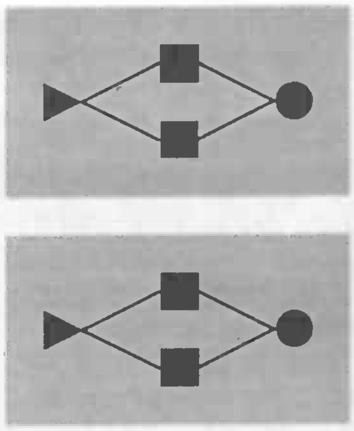


Figure 1.4: *Two network architectures for two subproblems*

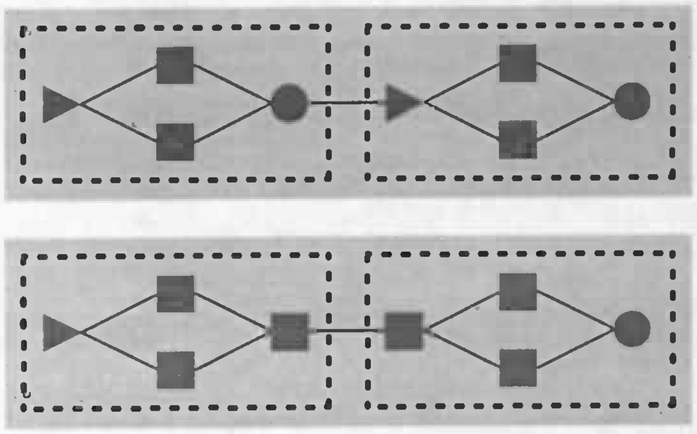


Figure 1.5: *(a) A network defined by two sub–networks,*
(b) The merged network

a new area of research. Using this method causes the need for neural networks that are built up hierarchically. A hierarchically built neural network is a network that consists of two or more subnetworks that are connected to each other. Each of the subnetworks solve a part of the main problem.

Suppose we have one problem which can be solved by two neural networks (Figure 1.4) using the divide and conquer method, by connecting the output signal of the first network (e.g. the license plate) to the input of the second network. Obtaining a solution to the whole problem, one hierarchical network, consisting of the two subnetworks can be created (Figure 1.5a). The subnetworks are surrounded by a dashed box, indicating the previous boundary of this structure. The input and output neurons of the two separate networks are still viewed as input and output neurons. Figure 1.5b shows the merged network. The output neuron of the first group and the input neuron of the second group are now considered to be hidden neurons. This is because there is no possibility to propagate the error back from an input neuron to an output neuron (like in Figure 1.5a).

1.3 Designing neural networks

Solving problems with a neural network can be done by creating a program or by using an already existing simulator. When simulating a neural network by writing a program, this program must initialize, calculate and evaluate a neural network. Routines to perform these tasks must be written by the user himself.

Existing simulators offer an instruction set to initialize, manipulate and simulate a neural network. The most important issues writing a simulator are the instruction set and the user interface. Some simulators can only be manipulated using a graphical user interface. Others offer a large instruction set but have no graphical user interface at all.

At the department of computing science at the Rijks*universiteit* Groningen, a simulator called InterAct (see [4]) is being developed and used. This simulator offers quite a large instruction set and some very useful observations to investigate the current status of the network. Observations are available to view the network status, structure, target and current output, error and more. The simulator does not offer the possibility to design a network interactively (where one can actually create and manipulate the objects by mouse) using a graphical user interface (GUI). InterAct currently provides two possibilities to design a network: by using a graphical user interface with menu–buttons or by writing a C–program and linking this with the InterAct library.

1.3.1 Using the graphical user interface of InterAct

One way to design a neural network in InterAct is to create the network using menu–buttons. To generate the neural network for learning the XOR–function as viewed in Figure 1.3, a user should take the following steps:

1. Define a new network (from the File–menu, select "New");
2. Create two input neurons (from the Edit–menu, select "Add Neurons", see figure 1.6);
3. Create two hidden neurons and one output neuron;
4. Create connections from the input neurons to the hidden neurons (from the Edit–menu select "Add Connections", see Figure 1.7);
5. Create connections from the hidden neurons to the output neurons.

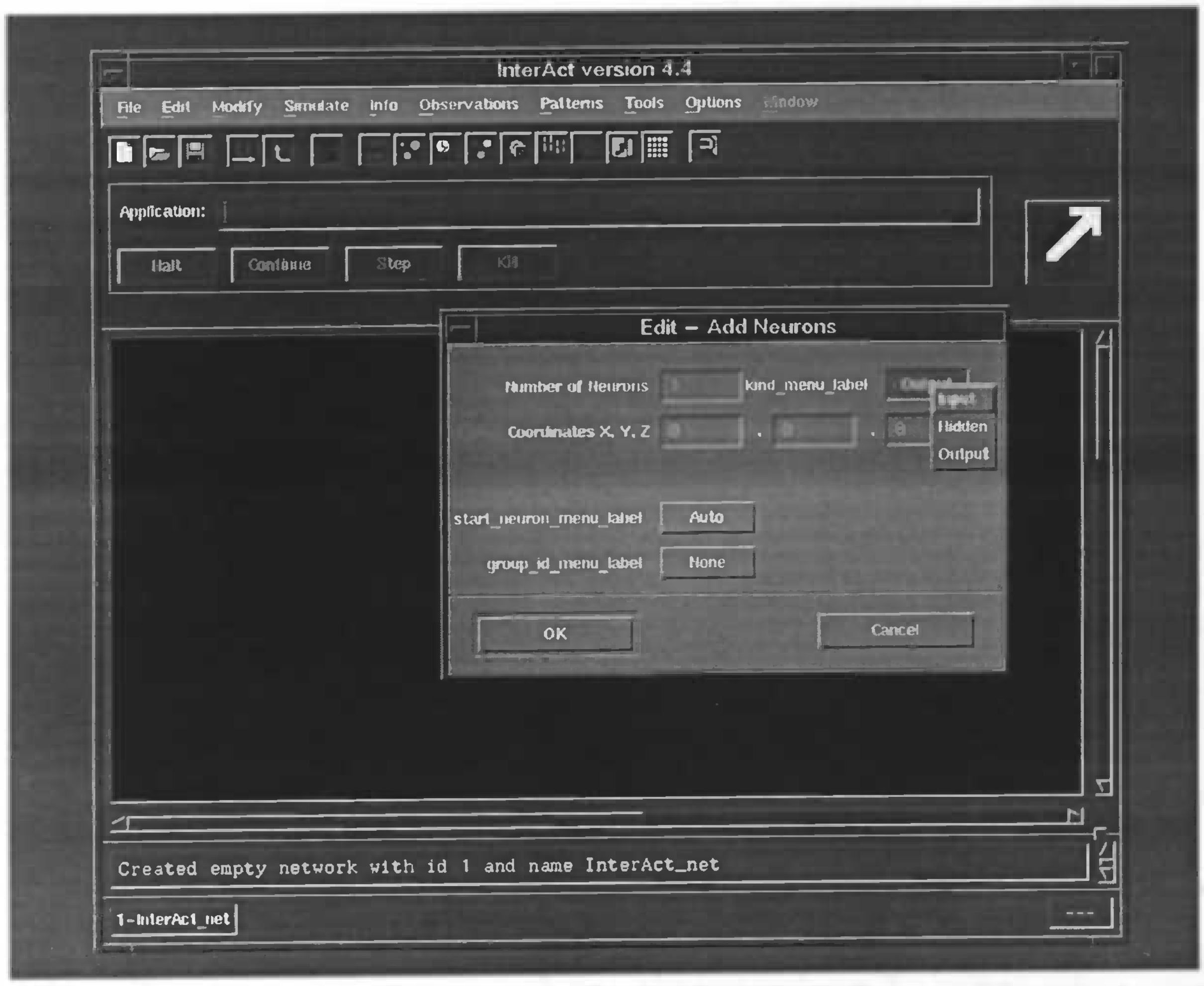
1.3.2 Using the InterAct library

Simulating with InterAct can also be done by creating a program, written in C, which uses the libraries of InterAct. InterAct offers the possibility to use calls inside a C–program to create and manipulate a network. Suppose `nr_inputs`, `nr_hiddens` and `nr_outputs` represent the number of input, hidden and output neurons, respectively. In that case the next lines of C–code will add `nr_inputs` input neurons, `nr_hiddens` hidden neurons and `nr_outputs` output neurons to an existing network:

```
/*add input neurons*/
default_Skind(kind_Sinput_neuron, &status);
neuron_Sadd(neuron_Sauto_t ,0L, 0L, 0L, 0L, nr_inputs, &dummy,
            &status);

/*add hidden neurons*/
default_Skind(kind_Shidden_neuron, &status);
neuron_Sadd(neuron_Sauto_t, 0L, 0L, 0L, 0L, nr_hiddens, &dummy,
            &status);

/*add output neurons*/
default_Skind(kind_Soutput_neuron, &status);
```



```
Created empty network with id 1 and name InterAct_net
```

Figure 1.6: *Screenshot of InterAct: creating neurons*

```
neuron_Sadd(neuron_Sauto_t, 0L, 0L, 0L, 0L, nr_outputs, &dummy,
            &status);
```

Normally, of course, the other parameters give more information about the neurons that have to be added (neuron ID, position, a return–value of the status). Also observations can be started by calls from the C–program to the InterAct library.

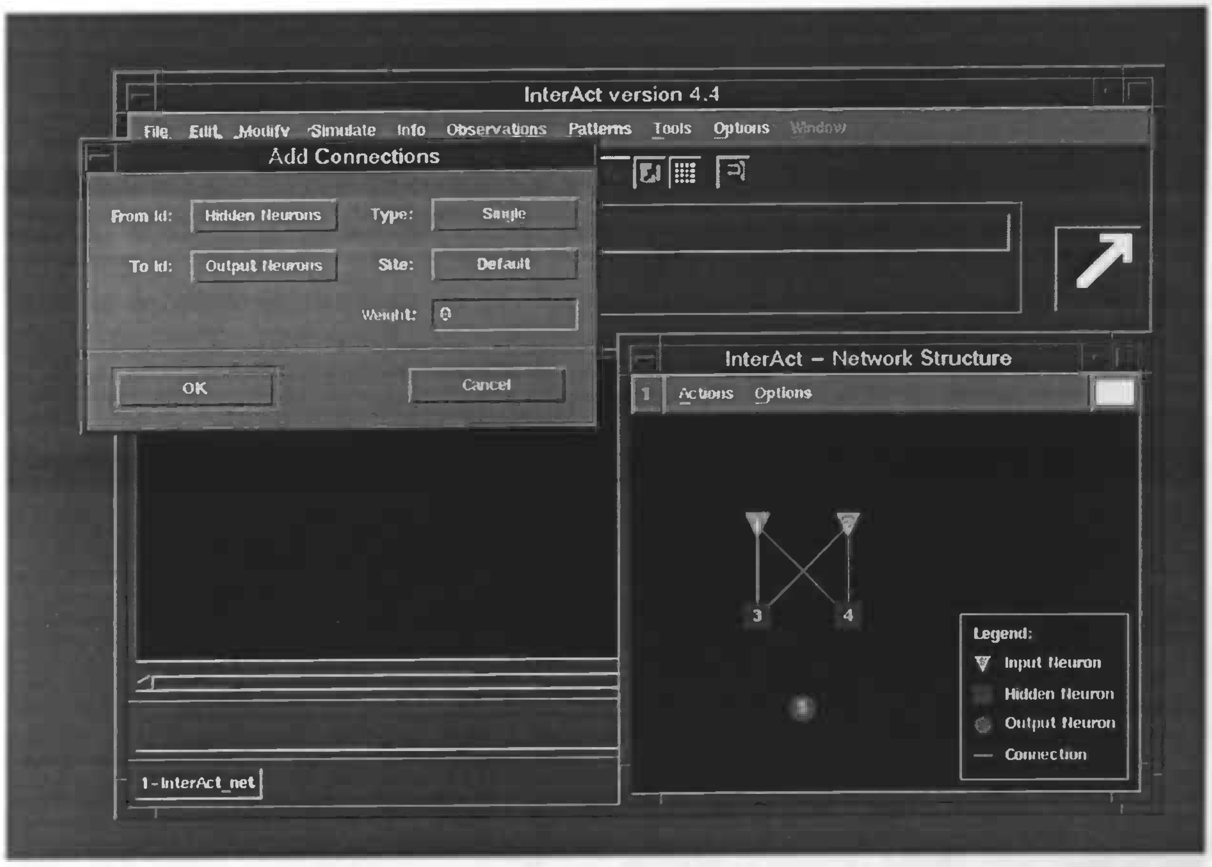
1.4 Goal

InterAct (see section 1.3) does not offer the possibility to create a network interactively using a graphical user interface (GUI). Recently the need to design a network using such an interface has become stronger. Also the possibility to define a hierarchical structure (see section 1.2) is very desirable.

The goal of this thesis is:

Designing a graphical user interface for interactively creating and manipulating hierarchically based neural networks

Chapter 2 analyses the goal. It will give an overview of the demands of the interface that has to be written. Furthermore some existing GUI's are investigated. In chapter 3 a functional descrip-



From Id: Hidden Neurons
Type: Single
To Id: Output Neurons
Site: Default
Weight: 0

Figure 1.7: *Screenshot of InterAct: creating connections*

tion is presented. Chapter 4 discusses the implementation of the user interface and chapter 5 gives an evaluation of the work done within this thesis.

Chapter 2 Problem analysis

In this chapter we will further analyze the requirements and demands on a graphical user interface in InterAct and observe some existing GUI's. First, section 2.1 discusses the demands on a graphical user interface in InterAct. Section 2.2 points out the observations on some existing graphical user interfaces and section 2.3 gives a conclusion.

2.1 Demands on a graphical user interface in InterAct

The requirements on the graphical user interface (or network editor) that has to be designed are listed below:

1. It should be possible both to view the current network structure and to obtain characteristics of specific elements of this structure.
2. Network structures have to be manipulable. When manipulating neural networks it is very useful to manipulate more elements at the same time.

As explained in 1.2, the possibility of creating subgroups in the network is also one of the requirements on the interface.

2.2 Related graphical user interfaces

Decisions about programming a graphical user interface are amongst other things, made by investigating other interfaces. Some existing simulators for neural networks as well as a drawing program are described below. The drawing program is observed because drawing programs also have to create and manipulate specific objects. While describing these programs, we investigate whether the programs satisfy the requirements listed in section 2.1.

2.2.1 NeuralNet

NeuralNet (see [5]), running on UNIX/X–Windows, is a simulator for neural networks which gives the possibility to open and save networks, patterns (the inputs and the target outputs) and weights (see Figure 2.1).

Visualization of the current network structure

The main window of NeuralNet views the current network structure (see Figure 2.1). Changes in the structure of the network will be visualized immediately in this window.

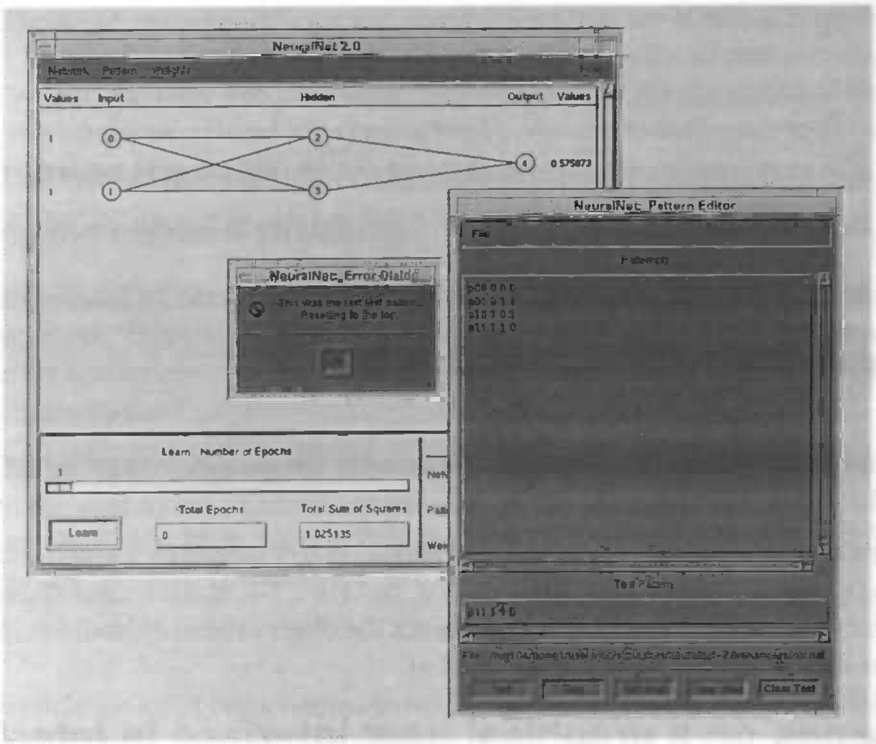


Figure 2.1: *Screenshot of NeuralNet*

NeuralNet offers the possibility to step through the patterns. As shown in the big popup–window of Figure 2.1, a set of input patterns with the belonging output patterns can be loaded and presented to the network. The resulting outputs of these input patterns are viewed in the main window at the input and output neurons.

Manipulating the network structure

When designing a network, the user is asked to define the number of inputs, the number of outputs, the number of hidden layers and the number of neurons per hidden layer. The maximum number of hidden layers is three.

NeuralNet offers two possible ways to create connections:

- Fully Connect: all neurons will be connected with the previous and the next layer.
- Neurons can also be connected interactively by using the mouse. Only connections to neurons in the next layer can be created. Defining the connections can be done by pressing the mouse four times. First the user has to click on the first neuron of the first layer from which the connections have to be created, then on the last neuron of the first layer; after this the first and last neuron of the receiving layer have to be pointed out.

Making connections using the mouse button is not functioning correctly. Creating a network as shown in Figure 2.2 is not possible. One should try to make connections from neurons 0–2 to 3 and then connections from 1–2 to 4. When defining these last connections the program gives the error (see small popup–window in Figure 2.1):

```
This node is already sending output to other node(s)
```

However, the possibility to create connections from more than one neuron to more than one neuron is a necessary feature for most neural networks.

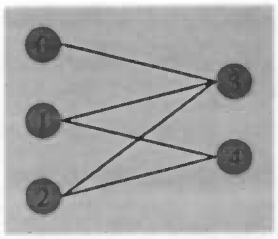


Figure 2.2: *Part of a neural network that can't be created by NeuralNet*

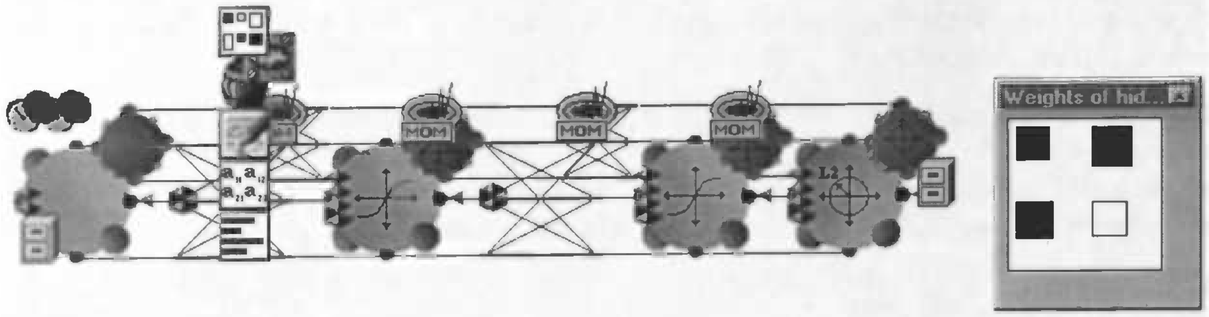


Figure 2.3: *Screenshot NeuroSolutions*

Conclusion

NeuralNet does not offer much tools for creating and manipulating a neural network. Because of these restricted possibilities the view of the network is very clear. Also the possibilities to manipulate the network structure are restricted. Making both single and multiple connections is a very useful feature provided by NeuralNet, though it is not functioning very good. NeuralNet does not support hierarchically structured networks.

2.2.2 NeuroSolutions

Visualization of the network structure

NeuroSolutions (see [6]) runs on DOS/Windows. It is a simulator for neural networks, which uses icons not only to visualize the status of the neural network but also to indicate which observations are running. In Figure 2.3 we can see the visualization of a Multi Layered Perceptron. Each of the bigger spheres is a member of the so called "Axon family". It represents a layer of processing elements. The first Axon component handles the input layer. There is a file component attached for reading the inputs.

The last component is used for the back propagation algorithm. It reads the desirder output (target) from file (another file component is used here) and the weights are adjusted (the little spheres attached to the bigger ones indicate back propagation planes). The two components in between are used to visualize the hidden layers. In this case, the "TanhAxon" is used. Many other members of the Axon family can be used, including SigmoidAxon, SoftMaxAxon, WinnerTakeAllAxon and GaussianAxon. The difference between these components is the function that the neuron uses to calculate the activation (output).

Between the input layer and the first hidden layer, a few observations are attached to the synapses. These observations are visualized as icons. NeuroSolutions offers several observations, including a matrix viewer, a matrix editor, a bar chart, a data editor and an image viewer.

On one hand the use of icons to indicate the observations that are running is quite good, but on the other hand, when a user wants to run several observations, with all that information attached to a network element, the view becomes quite chaotic. Another disadvantage of NeuroSolutions is the fact that the layers are viewed as separate icons, so there is no proper total view of the network like in NeuralNet (Figure 2.1).

Manipulating the network structure

Components are created by first choosing a component, after which the user can choose the position of the component. This procedure has to be repeated to create more components of the same type. Components can be connected using a "full connection" component (see Figure 2.3) or by user specified connections.

Perhaps it is a better idea to offer modes to design the network. Offering modes to design a neural network allows the user to create more components of the same type by just placing them after selecting the correct mode. Now the user has to choose a component to add and then place this component for each component that is needed in the network.

Conclusion

NeuroSolutions offers a lot of components to build a network. Also a lot of observations are supported by NeuroSolutions. Since the visualization of this network is very complete (much information is visualized in the main window), the total view of the structure becomes a chaos. Users who are not very familiar in creating neural networks will get difficulties in creating one. Hierarchically based networks are not supported by NeuroSolutions.

2.2.3 NCS NEUframe

Visualization of the network structure

NCS NEUframe (see [7]) runs on DOS/Windows. NCS NEUframe also uses icons to visualize the network's structure. In Figure 2.4, the status of a created network is viewed. After the first input–stream is decoded, it is presented to the actual neural network.

The network can receive five different input streams (see the left–side of Network 1 in Figure 2.4) and two output streams (see the right–side of Network 1). The input streams are the training inputs, the training targets, the test set inputs, the test set outputs and the query inputs. The outputs describe the training errors, the test set errors and the query outputs. The outputs in Figure 2.4 are attached to observations. The output streams can also be used as inputs for another neural network.

Manipulating the network structure

The network is created in the same way as with NeuroSolutions (section 2.2.2): by first defining which component (icon) has to be created and then defining the position where this component has to be located.

The structure of the neural network can be changed by filling in a form which defines the number of layers, the number of neurons in the current layer (this current layer can also be changed in the form) and the transfer function of the neurons. Figure 2.4 shows the popup–window that contains this form.

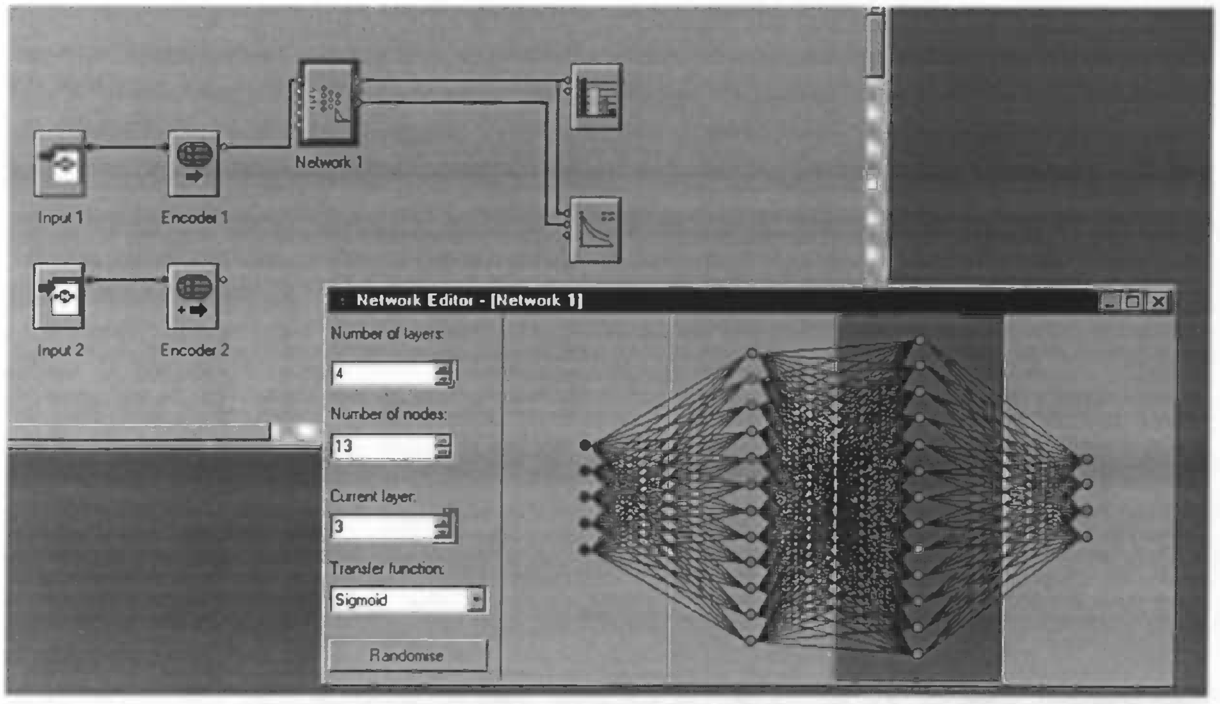


Figure 2.4: *Screenshot NCS NEUframe*

Conclusion

NCS NEUframe gives a clear view of the network structure. Components defining a neural network can be examined for their contents. Unexperienced users are able to create neural networks without very much trouble. More than one neural network can be defined and connected inside the total network, but a hierarchical structure is not possible.

2.2.4 SNNS

SNNS (see [8]) is a neural network simulator running on UNIX/X–Windows. It is probably the most common used simulator.

Visualization of the network structure

When SNNS is started a command window is created. From this window the current network structure can be obtained in a separate window (see Figure 2.5). Viewing the network can be done with or without the synapses between the neurons.

Manipulating the network structure

The use of this interface is not very intuitional. Networks can be loaded and saved using the interface. In the first lines of a saved network it is stated that the file is generated by SNNS. However, the possibility to change the network is not clearly described in the menu of the interface.

Another disadvantage is that instead of offering more submenus to manipulate the network, SNNS offers a few submenus in which many actions are possible (e.g. one menu to load and save everything: networks, patterns, result files, configuration and log files).

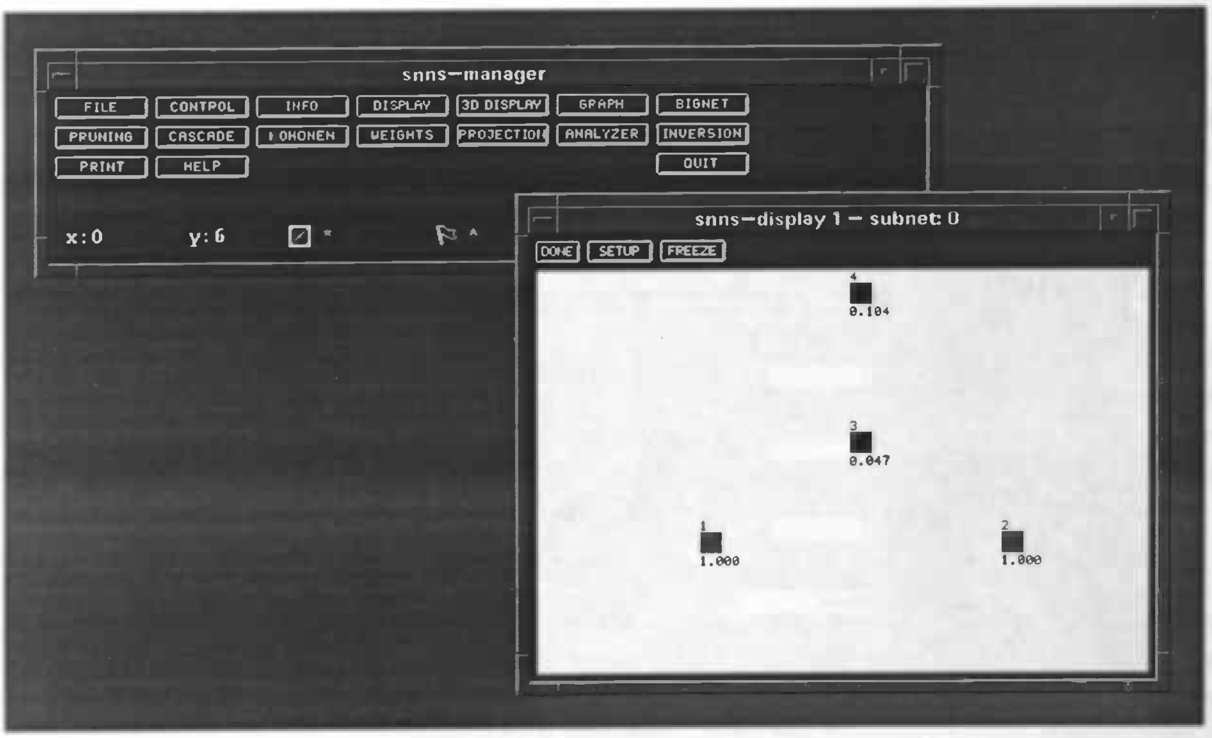


Figure 2.5: *Screenshot SNNS*

Conclusion

SNNS offers a lot of features for manipulating neural networks. However, without knowledge about neural networks or without a good reference manual it is not possible to manipulate the network structure using SNNS.

2.2.5 Interleaf 6

Interleaf 6 (see [9]) is a word processing program running under both DOS/Windows and UNIX/ X–Windows. This program is examined because of the possibility to work with a limited set of object types, which have to be created and manipulated. Because Interleaf 6 does not deal with neural networks, only the manipulation techniques of Interleaf 6 are discussed.

Interleaf provides a tool for drawing graphics (see Figure 2.6). Drawing graphics in Interleaf 6, some aspects of the graphics tool turned out to be very useful in a graphical user interface for ANNs. For instance selecting lines can only be done by pressing on the line itself (lines can be considered as connections). Pressing the "bounding box" (the rectangle the line is covering) is not sufficient.

The graphics toolbar provides several tools (see Figure 2.6): edit–mode (for moving, resizing, etc.), line–mode, arc–mode, etc. When pressing the mouse button only once on "line–mode" while in edit–mode, the mode will switch back to edit–mode after drawing a line. When double–pressing "line–mode", the mode will remain unchanged after drawing a line. Since the network editor which will be written for InterAct will probably need several modes, this might be an idea for the interface.

In Interleaf the user is able to select more than one object at a time. Selecting one object can be done by pressing the left mouse button on the object. Adding more objects to the list of selected

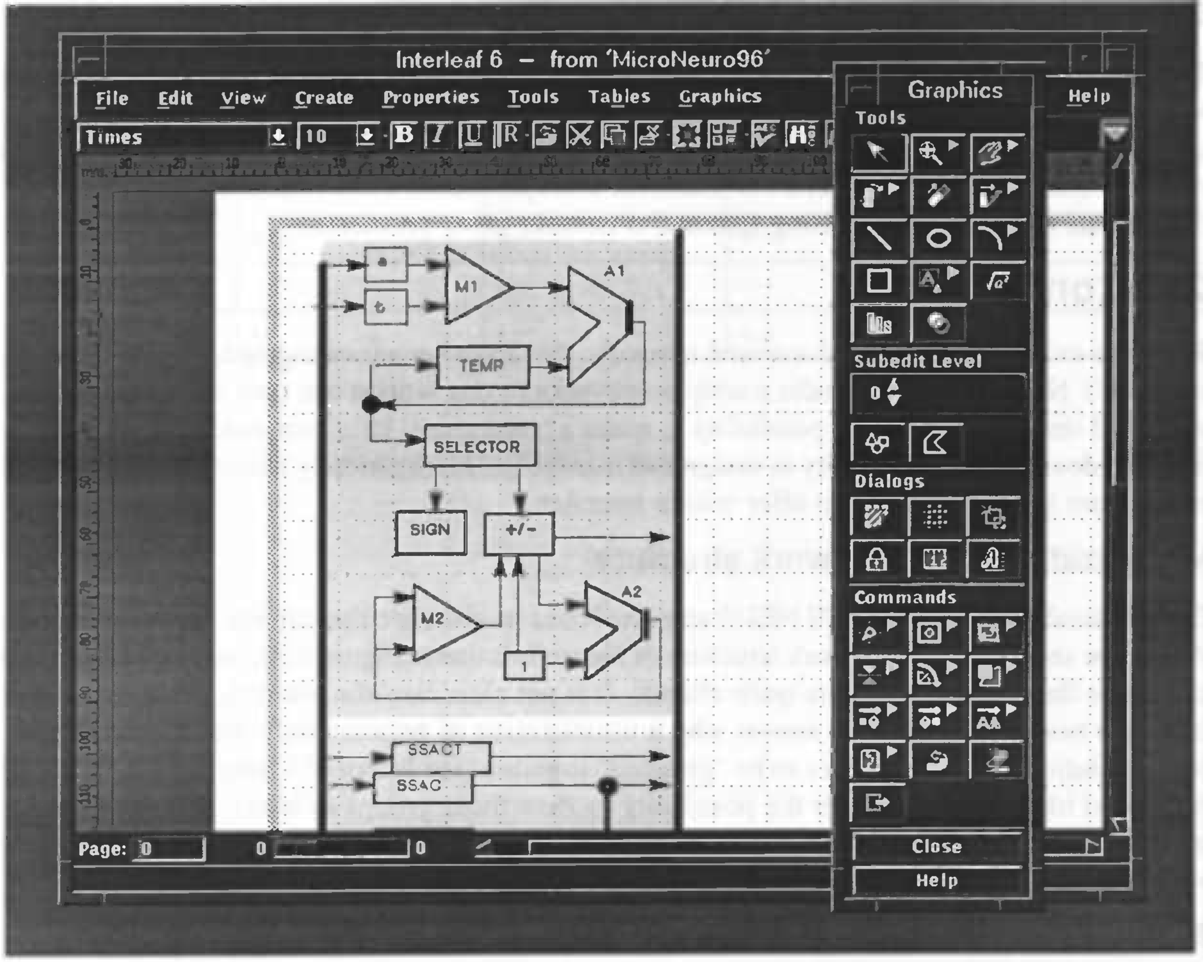


Figure 2.6: *Screenshot Interleaf 6 with graphics tool*

objects can be done by pressing the middle mouse button on the objects that have to be added to the list. This strategy makes it possible to manipulate more than one object at a time (e.g. moving / deleting more selected objects).

One disadvantage of Interleaf is the way objects can be copied. Once an object has been copied, the copy–buffer is empty. Copying an object several times can be done by selecting the object, copying it and repeating this procedure several times. A better way to copy an object could be by selecting an object and copying it several times, without selecting the object again (i.e. leaving the object in the copy–buffer).

The graphics tool of Interleaf 6 offers the possibility to "group" more objects together. When more objects are selected, these objects can be defined as a new object (a group). Manipulation features are now being applied on this new group, not its contents. Manipulating the contents of a group can be done by "un–grouping" the group or by "entering" the group. When double–clicking the left mouse button on a group, the edit level changes (the group is "entered") and contents of the group can be manipulated. Objects that are not inside the group are not manipulable anymore. The current edit level is displayed on the graphics tool bar window ("Subedit level").

Conclusion

Interleaf 6 is not very difficult to use for unexperienced users. The pixmaps, displayed on the buttons to edit a graphical arèa, show the meaning of the buttons clearly enough. the way of selecting connections is also very good. Interleaf 6 shows a way to design hierarchically. The way to manipulate this hierarchical structure is intuitive. Unexperienced users will understand the way to manipulate a graphical area using Interleaf 6 without many problems.

2.3 Conclusion

From the existing simulators for neural networks that have been investigated, we have seen that only NCS NEUframe can handle multiple networks in one workspace (see 2.2.3). None of the observed simulators offer the possibility to make a hierarchical structure inside the network. Interleaf 6 does provide the ability to design and manipulate hierarchically based components. This is a feature that we do want to offer within InterAct.

Visualization of the network structure

Both NeuroSolutions and NCS NEUframe use icons to visualize the current network structure. As can be seen from the network structure of NeuroSolutions (Figure 2.3), the use of icons only can make the network structure quite chaotic. It is not clear how the network is constructed out of single neurons. This is the reason why a visualization of neurons only would probably be a better solution. When neurons can be "grouped" together (see Interleaf 6, section 2.2.5), it could be a good idea to give the user the possibility to view these groups as icons. Different kinds of neurons can be visualized with different symbols, using the symbols already present in the observation for the network structure in InterAct (see Figure 1.7).

Manipulating the network structure

NeuralNet, NeuroSolutions and NCS NEUframe can make connections from one layer to another by using just one command. However, the possibility to create user specified connections is only offered by NeuroSolutions. In InterAct, we want to make both single connections and more connections at a time. Most of the time a user wants to fully connect one layer with another. This can be done by offering the possiblity to create multiple connections.

From the observed neural network simulators only NeuroSolutions and NCS NEUframe offer the possibility to actually place the objects that are created. A disadvantage of both packages is that each time one wants to create an object, the object type has to be selected first. Creating more objects of the same type has to be done by selecting an object type, placing the object and repeating this procedure. A solution to this problem would be to offer several "Tools" (see Interleaf, section 2.2.5) from which the user can choose the object that is going to be created, together with one edit–tool (for manipulating the network: moving, deleting, etc.). Choosing between these tools could be done by using the "double–click" method of Interleaf.

None of the observed simulators for neural networks provide the possibility to manipulate more objects together. Only Interleaf (section 2.2.5) offers this possibility by selecting more objects using the middle mouse button. Since not every mouse button has got a middle mouse button, it could be an idea to use the left mouse button while pressing the shift key (see also [10]).

Another feature that is not offered by the simulators that have been observed is the ability to create and manipulate hierarchically structures objects. Interleaf 6 does provide this feature. The way hierarchically based objects are created and manipulated in Interleaf 6 is very intuitive.

Evaluating the observed graphical user interfaces the following table can be created:

GUI	**Creation**	**Manipulation**	**Intuitivity**	**Hierarchically**
NeuralNet	+–	+–	+	––
NeuroSolutions	+–	+–	–	––
NCS NEUFrame	+–	+–	+–	––
SNNS	––	––	––	––
InterLeaf 6	+	++	++	++

Chapter 3 Functional description

In this chapter we will discuss the functional description of the graphical user interface for InterAct (from now on the term "network editor" will also be used). Section 3.1 gives a summary-of the different types of objects that can be created and section 3.2 gives an overview of the possible operations that can be applied on these objects.

3.1 Objects

The objects that are used in the network editor can be divided in two groups: leaf cells and composite cells. Where composite cells are composed of other cells, leaf cells are the basic network elements. Leaf cells form the basis of the network and can not be unfolded into other objects.

In this section, we will give a description of the possible objects of the neural network. All objects have several properties. Not all properties of an object are explained, only the most important properties.

3.1.1 Leaf cells

We will now consider the leaf cells. The network can consist of the following leaf cells: neurons, connections, fuzzy cells, algorithm cells and data cells.

Neurons

Neurons form the basis of a neural network. They have several properties. The most important properties of a neuron are: the neuron ID, the neuron type, the transfer function, the neuron sites and the connected synapses.

Each neuron has its own unique ID within the network. As already explained in chapter 1, there are three different types of neurons: input neurons, hidden neurons and output neurons. These can be visualized as respectively triangles, squares and circles, according to the standard of InterAct.

The transfer function of a neuron is the function which is calculated within a neuron. There are different functions possible. In section 2.2.2, we observed that NeuroSolutions offers different icons to visualize different transfer functions within a layer.

By defining sites, connected to a neuron, we create a tool for receiving the inputs. Explaining the process inside a neuron in section 1.1, we assumed that the incoming values at the synapses are

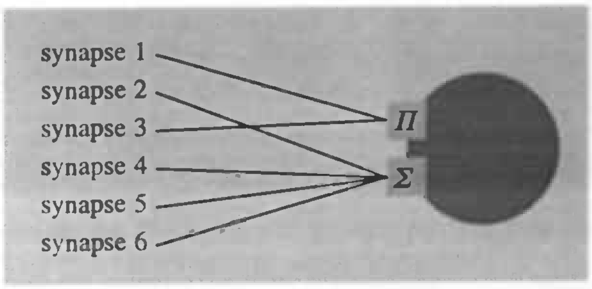


Figure 3.1: *Using neuron sites*

being added before presenting the result to the transfer function of the neuron. Mostly this is the case, but somtimes other calculations are made before presenting the input patterns to the transfer function. In Figure 3.1, a more detailed example of a neuron is presented. In this figure, the first and third input are being multiplied before presented to the transfer function. The other inputs are being summed. The processing units receiving these inputs before presenting the result to the transfer function are called *sites*.

Also the synapses are properties of a neuron. The synapses connect one neuron to another (or to be precisely to the next site).

Connections

The properties of connections (synapses) are the neuron presenting its output signal to the synapse, the neuron receiving the signal, the connection weight and the connection type. Since a synapse is connecting two neurons, there is always an input neuron and an output neuron. Also the weight is a property of a connection. The weight determines the weight factor used before the signal is presented to the sites.

The connection type describes the type of the connection. A synapse can be single connecting or bidirectional connecting. A bidirectional connecting synapse between two neurons can be considered as two synapses, one connecting the first neuron to the second neuron and one connecting the second neuron to the first neuron. The two neurons will act both as input neurons and as output neurons to such a connection.

Another property of connections is the connection state. Visualizing a network without the connections sometimes gives the user a better view over the network. The connection state indicates whether connections have to be drawn or not.

Fuzzy cells

Fuzzy logic is another method for solving non–linear problems. One of the advantages of using fuzzy logic is the fact that the rules defining a fuzzy algorithm are written by the user himself, so they are also better to understand for the user (it is not a "black box" like a neural network". More information about fuzzy logic can be found in [11].

Inside a network, it would be a good feature to connect fuzzy cells to neural cells. An example of this feature is a problem that can be solved with the "divide and conquer" algorithm explained in section 1.2.2. Suppose the input patterns of the problem could be "pre–processed" by using fuzzy logic, so better features can be offered to the network. In that case. the output of the fuzzy cell could be connected to the input layer of the neural network.

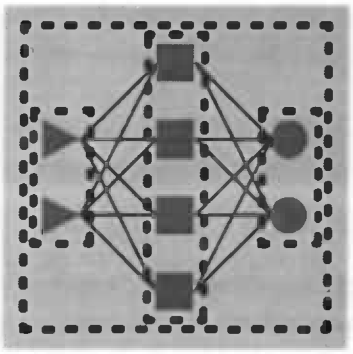


Figure 3.2: *A network with four homogeneous modules*

A fuzzy system has got its own hierarchy, which we will not explain here. Since the fuzzy cell is considered to be a leaf cell, we only deal with the properties position and connections. Other properties of a fuzzy cell are "hidden" inside the fuzzy system.

Algorithm cells

Algorithm cells are cells in which calculations are being made. Since input signals of neurons often have to be on the interval [−1, 1] and output signals on [0,1], one of the most simple examples of algorithm cells are the calculations that have to be made before presenting input patterns to a neural network and the calculations that have to made on output signals leaving the network.

Data cells

Presenting data to a neural network can be done by creating data cells. Data cells are cells which read data from a file (see the file icons in NeuroSolutions, section 2.2.2) or which contain matrices with data that can be presented to the network. Data cells can be used as inputs for a neural network or as target outputs of the neural network.

3.1.2 Composite cells

Unlike leaf cells, composite cells do not only have properties. Composite cells are cells that are made up of other cells. Where leaf cells do not have contents (or they can not be examined), composite cells can be "magnified" to examine or manipulate their contents.

Modules

Composite modules can be classified into two classes: *hybrid* modules and *homogeneous* modules. Hybrid modules contain different types of objects and homogeneous modules contain objects of the same kind. An example of a homogeneous module is an input layer that is "grouped" together. In NeuroSolutions (section 2.2.2), icons were used for each layer. This is an example of homogeneous modules. Hybrid modules contain different cells. An example of a hybrid module is a neural network in which more different neurons are grouped together.

In the remaining of this thesis the term "group" will also be used to indicate a composite module. In Figure 3.2 a network with several groups is drawn. Groups are visualized by dashed lines. All the four modules groups are homogeneous. One is containing input neurons, one output neurons, one hidden neurons and one is containing these three homogeneous modules. Visualizing groups by dashed lines will be used in the remaining of this thesis.

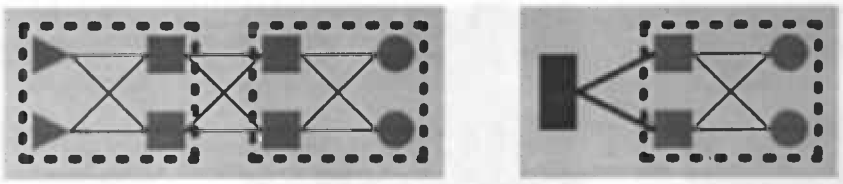


Figure 3.3: *The network divided in two groups: (a) The first group is opened, (b) The first group is closed*

Hierarchically built neural networks offer a way of information–hiding. Not all aspects of the neural network are important at one level but are dealt with in a lower level. This can be visualized by iconifying cells. In this way composite cells are visualized as black boxes without any information about their contents. Only by opening an icon the user is able to see the internal structure of a "closed" cell. In Figure 3.3 an example of a closed cell is illustrated. In the remaining of this thesis, the terms "opened group" and "closed group" will be used.

Specials

We would like to discuss some composite cells apart from the modules explained above. This is because those modules contain objects that have already been declared (leaf cells or modules). It is also possible to define composite cells that contain other objects. These special composite cells can not be edited using an editor for neural networks because other types of objects have to be edited. Examples of these special composite cells are fuzzy cells and statistical analyzers.

In section 3.1.1, fuzzy cells have been considered to be leaf cells. Fuzzy cells can also be considered to be composite cells. In this case it should be possible to "open" a fuzzy cell and edit its contents. In section 3.1.1, fuzzy cells were considered to be "black boxes".

Another possibility of special composite cells are statistical analyzers. These cells can be built out of more components to analyze data statistically.

3.2 Operations

In this section we discuss the operations that can be applied upon the objects described in section 3.1. Operations can be divided into two classes: structural operations and graphical operations.

3.2.1 Structural operations

Structural operations are operations that will change the internal structure of objects. We will discuss the following structural operations: creation, property change, elimination, merge and split.

Creation

For creating neurons we need the position of the neuron inside the network and the neuron kind. Newly created neurons receive a new unique neuron ID. Since a newly created neuron is not yet connected to any other neuron, the list of connections is still empty. It is possible to define a transfer function each time a neuron is created, but this takes a lot of the user's time. Another way of determining the transfer function is to create neurons with a default transfer function, which can be changed later.

Creating a connection requires two cells: the cell sending its output signal to the connection and the cell receiving the signal. The connection kind is not of importance, since bidirectional connections can be made by creating two single directional connections. All the leaf cells discussed in section 3.1.1, except for connections themselves, can be connected with each other. Data cells can be connected to neurons, neurons to algorithm cells, etc. Since composite modules (groups) can not receive input signals or send out output signals themselves, they can not be connected to other objects. However, groups are connected indirectly with other objects if their contents are connected to other objects. When one tries to make a connection to a group, there are some possibilities concerning this request. The request can be denied or connections can be created with the contents of the group. Creating connections to the contents a group can be done by making connections with all the leaf cells inside this group or by making connections to only the direct internal leaf cells of this group.

Creating a composite module requires a group of objects that become its children. One of the questions creating a composite module is whether this group of objects can be empty, so objects can be added afterwards. Also the position of the new group is a property that is necessary for creating a new group.

Another way of creating new elements is copying existing objects. Existing elements can be copied so new objects with the same properties will be created. Copying one connection does not seem to be very useful. Since connections are connected to two objects, copying this connection will create a new connection between these objects, which does not make sense. Another way to copy connections can be very useful. Suppose an object that is connected to another object is copied. When copying this object, it can be very useful to copy the connections that are attached to it also. Since the connections that are attached to the object are also properties of this object, making new connections attached to the objects that are copied would be a logical action.

Property change

It must be possible to change the properties of existing objects. Since each object has different properties, different menus have to be created to change these properties. An example of changing an object's properties is the change of the transfer function of a neuron. An overview of the properties that could be changed is given in section 3.1.

Elimination

After objects have been created, it must be possible to delete these objects. Elimination of objects has got a direct influence on the performance of the neural network (think about the effect of deleting a connection or a neuron).

Like in drawing programs (see also Interleaf, section 2.2.5), elimination could be done using a "cut"–operation. This operation will cut an object from the current workspace. Objects that have been cut can later be pasted. Pasting already cut objects implies the "re–creation" of these objects. New objects are created with the same properties as the objects that have been cut. Objects that have been cut be pasted at another position or in another group. Pasting an object to another group is done by cutting an object from one composite cell and pasting it in another composite cell.

Cutting an object that is connected to other objects will leave a connection in the network that is not attached at both sides. Since this makes no sense, it would be a good idea to remove that connection when the object is cut.

Objects that have been cut while connected to other objects can also be pasted. When pasting these objects, the connections can stay removed or new connections can be created, connecting

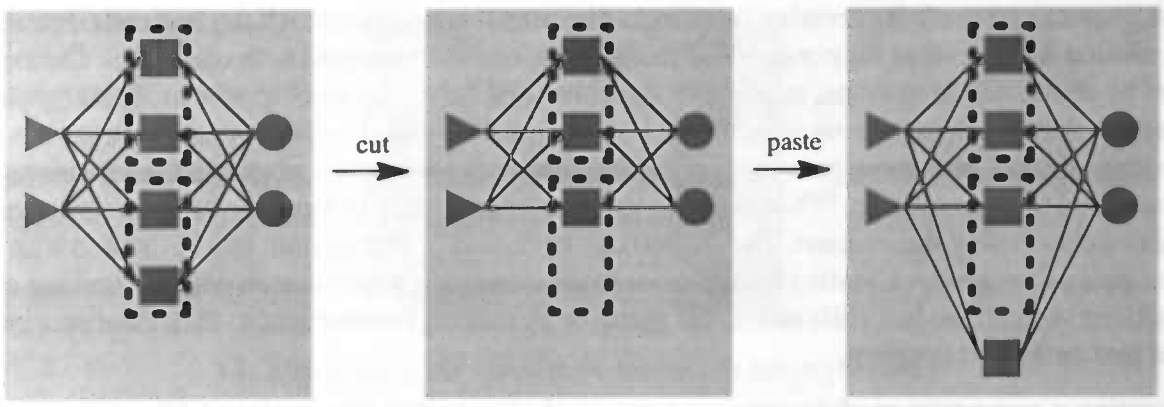


Figure 3.4: *Cutting a connected neuron and pasting it in another edit level*

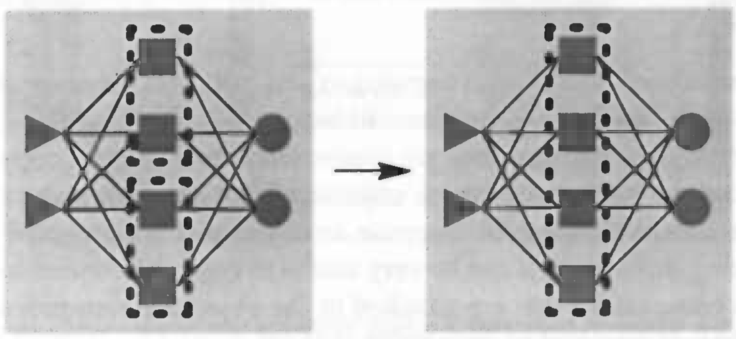


Figure 3.5: *Merging two composite cells*

the pasted object to the object it was connected before. Since connections are properties of the object, the latter proposal would be a good idea. In Figure 3.4, this proposal is visualized. In this figure one neuron, that was connected to other neurons, is cut from a composite cell and is pasted in another, "re–creating" the connections that have been pasted.

Merge

The merge operation is used for merging two composite cells into one. The number of neurons and connections in the network remains the same, but the hierarchical structure of the network changes. Because of this change in the hierarchical structure, the learning rules of the network will probably also change. Figure 3.5 gives a good example of merging two composite cells.

Split

The splitting of one composite cell is the inverse of the merge operation. Like merging, splitting does not change the number of neurons and connections in the network, but will change the hierarchical structure (see Figure 3.6).

3.2.2 Graphical operations

Where structural operations change the structure of the network, graphical operations only change the representation of the network. The internal structure is not affected by graphical operations.

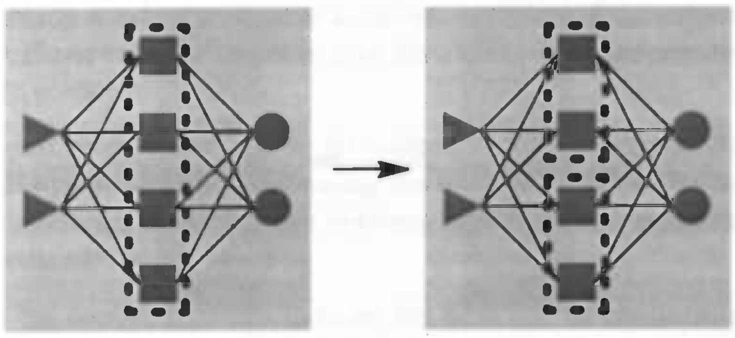


Figure 3.6: *Splitting a composite cell*

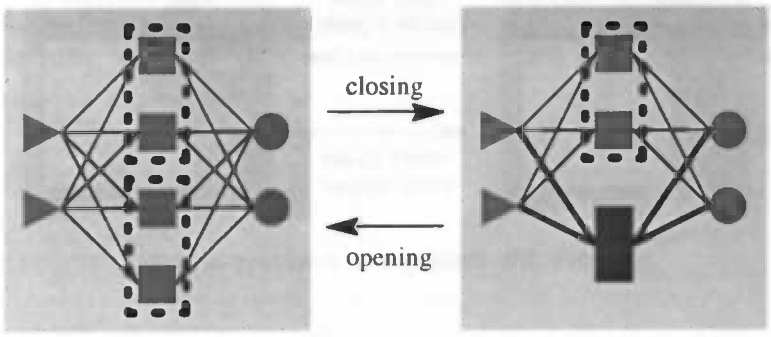


Figure 3.7: *Opening and closing of a composite cell*

Open / Close

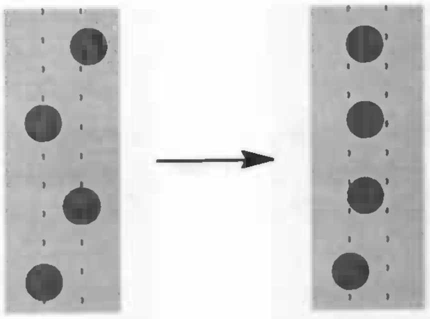
Hierarchically built neural networks offer a way of information–hiding. Not all aspects of the neural network are important at one level but are dealt with in a lower level. This can be visualized by iconifying cells. In this way composite cells are visualized as black boxes without any information about their contents. Only by opening an icon the user is able to see the internal structure of a "closed" cell. Figure 3.7 gives an example of opening and closing cells. In this figure, the synapses connected to a closed cell are drawn thicker to indicate that more synapses are connected.

Move

The user must be able to move certain elements to other positions. Moving elements is only done for visualization purposes. The internal structure of the neural network must remain unchanged.

Since connections are connecting other cells, connections can be moved by moving the cells that are connected to them. Moving a composite cell (opened or closed) must imply the movement of its contents. Suppose that one opens a closed group that has been moved. The contents of that cell are supposed to be moved too.

Besides of just moving elements to another position, certain operations must make it possible to move more elements to positions that are depending on the current positions of these elements. (alignment operations). It must, for example, be possible to move more elements to their own relative center position. Figure 3.8 shows the result of aligning the upper three elements to their relative vertical center position. Grid lines have been used to make this operation more clear.


3.8: *Alignment of the upper three neurons to left–right center*

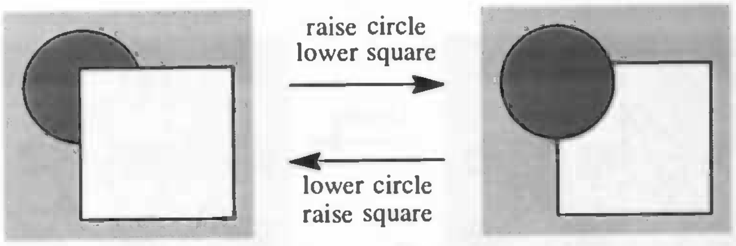


Figure 3.9: *Raising and lowering objects*

Resize

Groups have to be resizable. Resizing groupss make it possible to make a group bigger so new objects can be added to that group without the problem that objects will overlap. This operation only makes sense for groups, because resizing leaf cells would not result in a clear view of the network (one can think about input neurons that have different sizes).

Lower / Raise

Two possible operations to change the visibility of an object are:

- Lower: an object will be put behind all other objects. By changing the order of drawing all the objects, this object will be drawn before other objects. The advantage of this operation can be clearly seen when two objects are overlapping. The object that is lowered will be visible behind the other object.
- Raise: This operation is the reverse of lower. By raising an object, this object will be the last object that is drawn.

The routine of raising and lowering objects is also used in window managers. Windows that are covering other windows can be lowered, so other windows become visible. Figure 3.9 views the difference in visualization of raising and lowering a square and circle.

Information

It must be possible to obtain information about the properties of objects. This information can be visualized in the working window of the network editor (e.g. the neuron kind by different visualizations) or it can be presented to the user by popup windows.

For designing the network editor, it is necessary to find a good mixture between visualization of properties in the working window and presenting them using popup windows. Visualizing all properties in the working window will cause a chaotic overview of the network (see NeuroSolutions, section 2.2.2). However we do want to give the user as much information as possible without asking for popup menus.

Neurons can be visualized with the neuron ID written in it. Another possibility is drawing the neurons with a small symbol inside it indicating the transfer function. By drawing the neurons with three different symbols, the difference between input neurons, output neurons and hidden neurons can be visualized.

Drawing arrows on the connections can indicate the direction of connections. Another way of giving information about connections is visualizing the weight of a connection. This can be done by drawing a connection in different colors.

Closed groups can, just like neurons, be visualized with the ID written in it. Displaying ID's in opened groups is an idea, but could give a chaotic view when visualizing groups within groups (ID's will overlap).

properties of the building components [illegible]
them, section 2.2.3). Moreover [illegible]
the mixing for group [illegible]

[illegible]

Chapter 4 Implementation

In this chapter we discuss the implementation of the network editor in InterAct. The editor is written in C, using utility calls to an existing library called Ixm. The Ixm library is developed for InterAct and uses the Motif 1.2 library (see [12]). Motif is based on Xt, which is based on X–Windows (see [13]).

The Ixm library offers functions to create and control a graphical area. Before using other Ixm calls, Ixm has to be initialized. When Ixm is initialized, colors, fonts and cursors are loaded. Using these colors, fonts and cursors results in a standard view of the user interface. Different parts of the user interface of InterAct look the same using the Ixm library.

In the current version of the network editor, fuzzy cells, data cells, calculation cells and special composite cells are not implemented yet, because of a lack of time. Ideas about implementing these cells are listed in chapter 5. Also the network editor has not been linked to the InterAct library yet. Manipulation of objects only cause a change in the data structure within the network editor, the network used in InterAct remains unchanged. This is the reason why the ID's of objects in the network editor are not the same ID's that InterAct would use. The ID's that are used now are temporary and created by the network editor to see if the visualization is correct.

Chapter 6 will give an overview of the total source code that has been developed during this project. This chapter is divided into three sections. The first section discusses the data structure that is defined for the network editor. We first discuss this data structure because all routines in the network editor are using calls to create, examine and manipulate this structure. The second section discusses the visualization of the several objects of the neural network. In the last section we describe the manipulations that are possible in the network editor.

4.1 Data structure

In this section the data structure is discussed. This data structure is defined in `type.h`. In section 6.8 the total listing of `type.h` and `type.c` is presented. All objects in the network are implemented by **`Element`**. The module `type` supplies calls to create objects and to examine and change their properties inside the data structure.

4.1.1 Creation

Calls to create objects are listed below:

```
Element *CreateNeuron(parent, x, y, type)
Element *CreateConnection(from, to, state)
Element *CreateGroup(parent, x, y, width, height, opened,
                     able_opening, state)

    Element *parent, *from, *to;
    NeuronType type;   /* INPUT_NEURON, HIDDEN_NEURON,
                          OUTPUT_NEURON */
    Position x, y;
    Dimension width, height;
    ConnectionState state;   /* VISIBLE, INVISIBLE,
                                DEFAULT, PARENT */
    Boolean opened, able_opening;
```

Since neurons can also be created inside existing groups, the parent of the neuron (a group, which can also be the main network, which is in fact a composite module) is one of the properties given when creating a neuron. Apart from the coordinates of a neuron also the neuron type is a parameter of `CreateNeuron()`.

When creating a connection, the cells that are being connected have to be given to the function as parameters, as well as the connection state. The connection state can be visible, invisible, default or parent dependent. The values **VISIBLE** and **INVISIBLE** define whether we have to show or hide the connection. When the connection state is **DEFAULT**, the connection state is defined by a toggle in the main window of the network editor. If the connection state is **PARENT**, it is defined by the connection state of the parent. In this case the parent of the cell that is sending its output to the connection (this is always a group) is examined to determine what its connection state is, etc. This option makes it possible to view connections in one part of the network, but not in another.

The parameters of `CreateGroup()` determine the group's parent (this can be another composite module or the main network), the group's position and dimensions, a flag indicating whether the group is opened or closed, a flag indicating if the group can be opened and the connection state of the group. Since this function creates a new group and hence the group is still empty, there is no parameter defining its contents.

We already mentioned that the main network itself is considered to be a group. Since the main network consists of other objects, this is not in violation with the definition of a composite module, given in 3.1.2.

4.1.2 Examination of elements

In this section we discuss the calls to examine the different elements of the network editor. Calls have been divided in general calls (for all element types), calls for neurons, calls for connections and calls for groups.

General calls

General calls to examine elements of the network editor are listed below:

```
int GetElementID(elt)
ElementType  GetElementType(elt)
Position  GetElementXPosition(elt)
Position  GetElementYPosition(elt)
Dimension  GetElementWidth(elt)
Dimension  GetElementHeight(elt)
Element  *GetElementParent(elt)

    Element *elt;
```

where ElementType is defined as:

```
typedef enum {
    CONNECTION_TYPE, GROUP_TYPE, NEURON_TYPE,
    FUZZY_TYPE, CALCULATION_TYPE } ElementType;
```

The ID of an element can be obtained using `GetElementID()`. As already mentioned in the introduction of this chapter, this ID is only used inside the network editor for now. Whenever a new element is created, it receives a unique ID.

`GetElementType()` returns the type of the element. The element type is defined by `ElementType`. In `ElementType` `FUZZY_TYPE` and `CALCULATION_TYPE` have already been defined for future use of the network editor.

The rest of the calls should not be very difficult to understand when one looks at the names of the calls. The call `GetElementParent()` returns the parent of an element. For connections, the cell supplying its output to the connection is returned. The parent of any other element is always a group. The parent of the main group will be discussed later.

Calls for neurons

Calls to examine neurons are listed below:

```
int  GetNeuronNumConnections(neuron)
Element  *GetNeuronConnection(neuron, i)
Element  *GetNeuronParent(neuron)

    Element *neuron;
    int i;  /* i must be in [0 .. numconnections - 1] */
```

`GetNeuronNumConnections()` returns the number of connections connected to a neuron. These connections can be both outgoing and incoming. The call `GetNeuronConnection()` returns a connection of a neuron. When the integer `i` exceeds one of the boundaries given above, `NULL` is returned. The call `GetNeuronParent()` is equal to `GetElementParent()`.

Calls for connections

Connections can be examined using the following calls:

```
Element  *GetConnectionFrom(connection)
Element  *GetConnectionTo(connection)
Position  GetConnectionFromXPosition(connection)
Position  GetConnectionFromYPosition(connection)
Position  GetConnectionToXPosition(connection)
Position  GetConnectionToYPosition(connection)
ConnectionState  GetConnectionState(connection)

    Element *connection;
```

The first two calls return respectively the cell sending output to the connection and the neuron obtaining output from the connection. The next four calls return the position of the begin of the connection and the end of the connection. These positions are depending on the cells to which the connection is connected. The last call returns the state of a connection (using the return values described in section 4.1.1).

Calls for groups

The module `type` also provides calls to examine groups:

```
Element  GetGroupParent(group)
int  GetGroupNumInternals(group)
Element  *GetGroupInternal(group, i)
Boolean  GetGroupOpened(group)
Boolean  GetGroupAbleOpening(group)
ConnectionState  GetGroupConnectionState(group)
Boolean  GetGroupMainGroup(group)

    Element *group;
    int i;  /* i must be in [0 .. numinternals - 1] */
```

The first call returns the parent of a group. The second returns the number of internal elements inside the group. The third call can be used to retrieve an internal of a group. Requests of obtaining internal cells with indices outside the boundaries return `NULL`. The names of the next three calls should say enough about their purpose.

As already mentioned, the main network is implemented as a composite cell. To determine whether a group is in fact the main network, the call `GetGroupMainGroup()` can be used. If the group is the main network, the result of this call will be `True`, if not the result will be `False`.

Overview

Figure 4.1 gives an overview of the calls that have been discussed in this section. In Figure 4.2 we can see a very simple network structure and its hierarchical representation.

4.1.3 Manipulating elements

Calls to change elements in the network are also provided by `type` (section 6.1). Most of the calls have names that are analogue to the names for examinating the elements (e.g. `GetElementPar-ent()` and `SetElementParent()`). Calls have been divided into general calls, neuron calls, connection calls and group calls.

General calls

General calls are used to change the properties of an element. General calls are:

```
void  SetElementXPosition(elt, x)
void  SetElementYPosition(elt, y)
void  SetElementWidth(elt, width)
void  SetElementHeight(elt, height)
void  SetElementParent(elt, parent)
void  DestroyElement(elt)

    Element *elt, *parent;
    Position x, y;
    Dimension width, height;
```

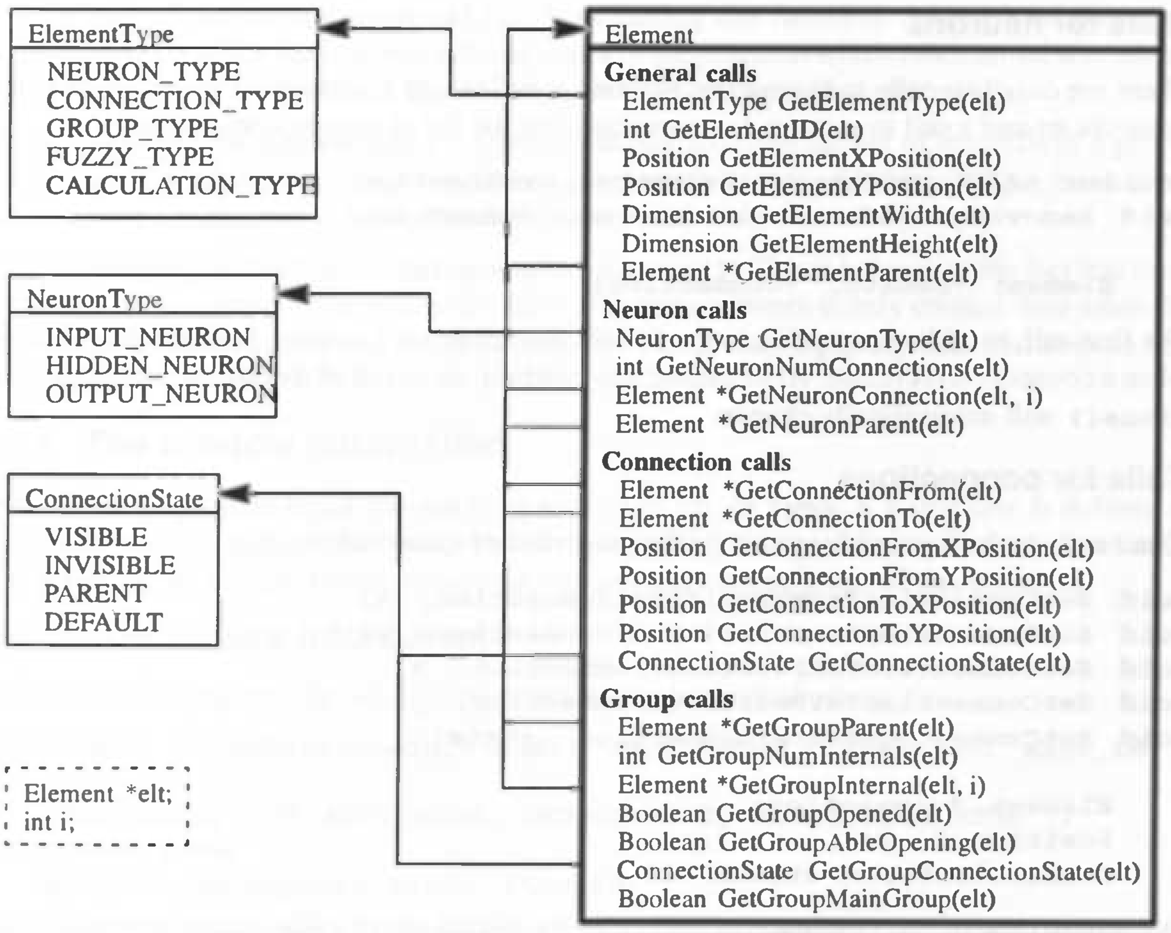

ElementType
NEURON_TYPE
CONNECTION_TYPE
GROUP_TYPE
FUZZY_TYPE
CALCULATION_TYPE
NeuronType
INPUT_NEURON
HIDDEN_NEURON
OUTPUT_NEURON
ConnectionState
VISIBLE
INVISIBLE
PARENT
DEFAULT
Element *elt;
int i;
Element
General calls
ElementType GetElementType(elt)
int GetElementID(elt)
Position GetElementXPosition(elt)
Position GetElementYPosition(elt)
Dimension GetElementWidth(elt)
Dimension GetElementHeight(elt)
Element *GetElementParent(elt)
Neuron calls
NeuronType GetNeuronType(elt)
int GetNeuronNumConnections(elt)
Element *GetNeuronConnection(elt, i)
Element *GetNeuronParent(elt)
Connection calls
Element *GetConnectionFrom(elt)
Element *GetConnectionTo(elt)
Position GetConnectionFromXPosition(elt)
Position GetConnectionFromYPosition(elt)
Position GetConnectionToXPosition(elt)
Position GetConnectionToYPosition(elt)
ConnectionState GetConnectionState(elt)
Group calls
Element *GetGroupParent(elt)
int GetGroupNumInternals(elt)
Element *GetGroupInternal(elt, i)
Boolean GetGroupOpened(elt)
Boolean GetGroupAbleOpening(elt)
ConnectionState GetGroupConnectionState(elt)
Boolean GetGroupMainGroup(elt)

Figure 4.1: *Relations between the calls to examine the data structure of the network editor of InterAct*

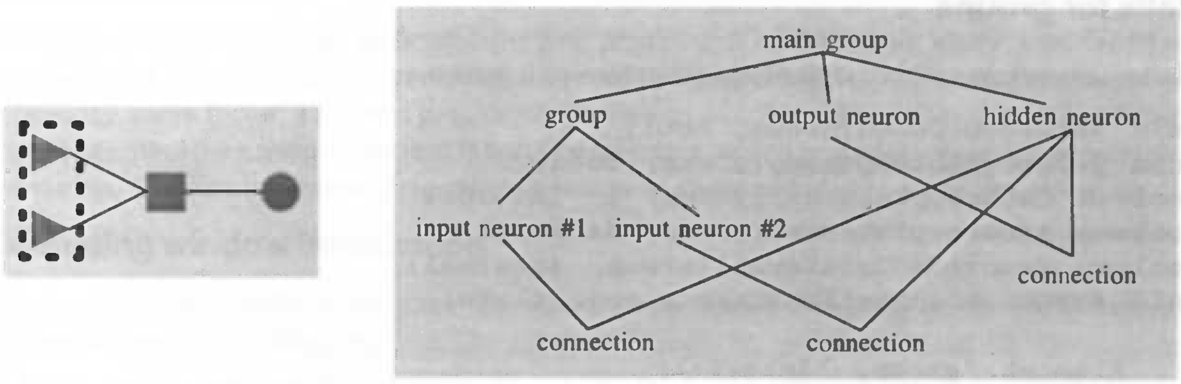


Figure 4.2: *A simple neural network and its hierarchical representation*

When we compare the calls to examine elements with the calls to change elements, we notice that there were two calls to examine an element which are not defined to change an element. This is the call to examine an ID and to examine the type of an element. There is no call to change the ID of an element because the ID of an element is a unique number which is defined by creation and which can not be changed. The type of an element can not be changed either. The call `DestroyElement()` destroys an element and frees the memory that was used for this element.

Calls for neurons

There are only two calls to manipulate neurons: a call to add a connection to the list of neuron connections and a call to remove a connection from the list of neuron connections:

```
Boolean AddNeuronConnection(neuron, connection)
void RemoveNeuronConnection(neuron, connection)

    Element *neuron, *connection;
```

The first call, to add a connection to the list of connections of a neuron, is automatically called when a connection is created. After calling this function, the result of `GetNeuronNumConnec - tions()` will automatically change.

Calls for connections

These calls make it possible to change the properties of a connection:

```
void SetConnectionFromXPosition(connection, x)
void SetConnectionFromYPosition(connection, y)
void SetConnectionToXPosition(connection, x)
void SetConnectionToYPosition(connection, y)
void SetConnectionState(connection, state)

    Element *connection;
    Position x, y;
    ConnectionState state;
```

The calls to change the coordinates of the beginning and the end of a connection have to be called after a neuron has moved. Their parameters are the connection and the new position. The last call is used to change the connection state.

Calls for groups

These calls can be used to change the properties of a group:

```
void SetGroupOpened(group, bool)
void SetGroupAbleOpening(group, bool)
Boolean SetGroupInternal(group, i, internal)
Boolean AddGroupInternal(group, internal)
Boolean RemoveGroupInternal(group, internal)
void SetGroupConnectionState(group, state)

    Element *group, *internal;
    Boolean bool;
    int i;  /* i must be in [0 .. numinternals] */
    ConnectionState state;
```

The first calls is used to make a group opened/closed. The `GetGroupAbleOpening()` call defines whether it is possible to open a group. Sometimes it could be useful to forbid the opening of a group.

The third call defines a new internal of a group. When the index `i` is equal to the number of internals, the internal is added to the list of internals. Another way to add a new internal to a group is using the call `AddGroupInternal()`. After calling this function, the result of `GetElementParent()` of the internal will automatically be the parent to which this internal was added. The return value of these calls is `True` if the internal has been added succesfully.

The call `RemoveGroupInternal()` removes an internal from the list of internals of a group. If the internal is found, it is deleted and `True` is returned. In case of failure, `False` is returned. `SetConnectionState()` is used to change the connection state of a group.

There is no analogue function for `GetGroupMainGroup()`. This is because of the fact that there is only one main network. The group describing the main network is only created once when the network editor is started and can not be deleted. Nor can an existing group become the main network.

4.1.4 The window information

One part of the module `type` has not been explained yet. In `type.h` a structure is defined in which all information about the network editor is defined. This structure is called `WindowInfo`. In this section we discuss the creation, examination and manipulation of this structure.

Creation of window information

The structure containing the window information can be created using:

```
WindowInfo  *CreateWindowInfo(mode,  zoom,  show_connection,  show_id)

    Mode mode;   /* EDIT_MODE, NEURON_MODE, CONNECTION_MODE */
    float zoom;
    Boolean show_connection, show_id;
```

In section 2.3, we suggested to define several "tools" like in Interleaf 6 (see 2.2.5) to add objects to the network and to manipulate these objects. We have chosen for three modes in the network editor: manipulation (edit) mode, neuron mode and connection mode. The usage of these modes is explained later in this chapter.

Defining a new structure can be done using four parameters: `mode`, `zoom`, `show_connection` and `show_id`. The first parameter defines the mode in which the editor will be started, the second defines the zoom factor. The third parameter determines whether connections that have a default connection state (or a connection that is parent dependent, which parent is parent dependent, etc.) are visible. The last parameter determines whether to show or hide the ID of the elements.

Examining window information

The most important calls to examine the window information are listed below. Other calls are explained later in this chapter.

```
Widget  GetWindowInfoDrawingArea(info)
Element  *GetWindowInfoMainGroup(info)
Mode  GetWindowInfoMode(info)
Boolean  GetWindowInfoShowConnection(info)
Boolean  GetWindowInfoShowID(info)

    WindowInfo  *info;
```

The first call is used to get the drawing area in which the network is drawn. More information about widgets can be found in [13]. `GetWindowInfoMainGroup()` returns the `Element` pointer to the group that is describing the main network. When `CreateWindowInfo()` is called, besides from defining a drawing area, the main network is defined. This is the group that will return `True` if `GetGroupMainGroup()` is called.

The `GetInfoMode()` call is used to get the current mode (edit, neuron or connection). The last two calls are used to see whether connections and ID's are visible.

The next function has not been discussed yet:

```
WindowInfo  *GetElementInfo(elt)

    Element *group, *elt;
```

This call is available for obtaining the window information of an element.

Manipulation window information

The following calls are available to manipulate the window information:

```
void  SetWindowInfoDrawingArea(info, w)
void  SetWindowInfoMode(info, mode)
void  SetWindowInfoShowConnection(info, bool)
void  SetWindowInfoShowID(info, bool)

    WindowInfo  *info;
    Widget w;
    Boolean bool;
```

Besides these calls there are more calls to manipulate the window information. Other calls are discussed later in this chapter. The names of the calls should say enough about their function. Figure 4.3 gives the total overview of the calls that have been discussed in this section.

4.2 Visualization of the network structure

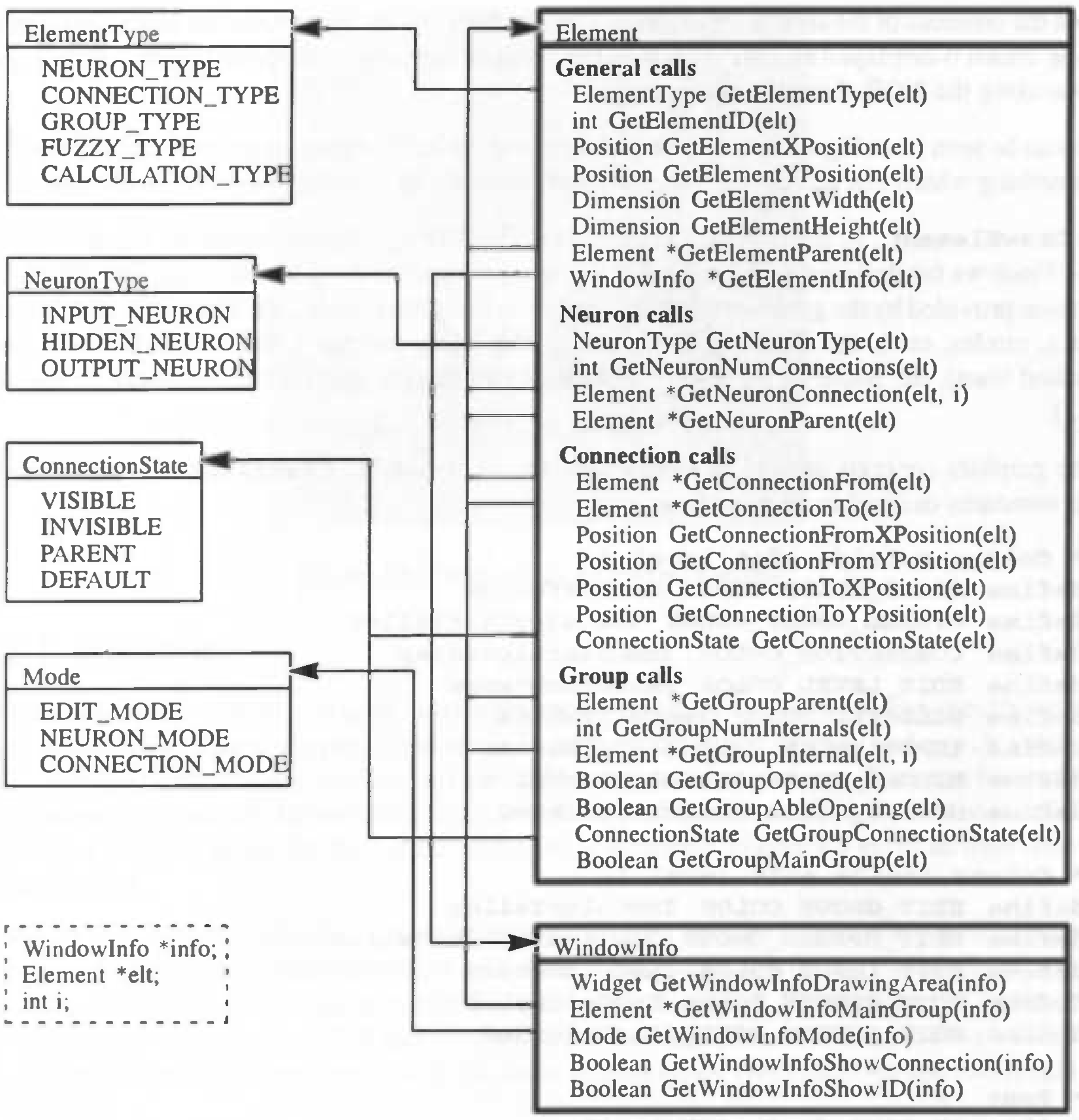
In this section we will discuss the visualization of the network by the network editor. Each of the objects mentioned in section 3.1 are visualized in different ways. In `visualize.c` (section 6.9) all elements are being visualized by the function `DrawElement()`. The syntax of `DrawElement()` is:

```
void DrawElement(elt, use_xor)

    Element *elt;
    Boolean use_xor;
```

The second argument is used to indicate whether the element has to be drawn using the normal drawing function or by using the XOR–function. The relationships between the inputs and the output of the XOR–function are:

Input #1	Input #2	Output
0	0	0
0	1	1
1	0	1
1	1	0



ElementType
NEURON_TYPE
CONNECTION_TYPE
GROUP_TYPE
FUZZY_TYPE
CALCULATION_TYPE
NeuronType
INPUT_NEURON
HIDDEN_NEURON
OUTPUT_NEURON
ConnectionState
VISIBLE
INVISIBLE
PARENT
DEFAULT
Mode
EDIT_MODE
NEURON_MODE
CONNECTION_MODE
WindowInfo *info;
Element *elt;
int i;
Element
General calls
ElementType GetElementType(elt)
int GetElementID(elt)
Position GetElementXPosition(elt)
Position GetElementYPosition(elt)
Dimension GetElementWidth(elt)
Dimension GetElementHeight(elt)
Element *GetElementParent(elt)
WindowInfo *GetElementInfo(elt)
Neuron calls
NeuronType GetNeuronType(elt)
int GetNeuronNumConnections(elt)
Element *GetNeuronConnection(elt, i)
Element *GetNeuronParent(elt)
Connection calls
Element *GetConnectionFrom(elt)
Element *GetConnectionTo(elt)
Position GetConnectionFromXPosition(elt)
Position GetConnectionFromYPosition(elt)
Position GetConnectionToXPosition(elt)
Position GetConnectionToYPosition(elt)
ConnectionState GetConnectionState(elt)
Group calls
Element *GetGroupParent(elt)
int GetGroupNumInternals(elt)
Element *GetGroupInternal(elt, i)
Boolean GetGroupOpened(elt)
Boolean GetGroupAbleOpening(elt)
ConnectionState GetGroupConnectionState(elt)
Boolean GetGroupMainGroup(elt)
WindowInfo
Widget GetWindowInfoDrawingArea(info)
Element *GetWindowInfoMainGroup(info)
Mode GetWindowInfoMode(info)
Boolean GetWindowInfoShowConnection(info)
Boolean GetWindowInfoShowID(info)

Figure 4.3: *Relations between the calls to examine the data structure of the network editor of InterAct*

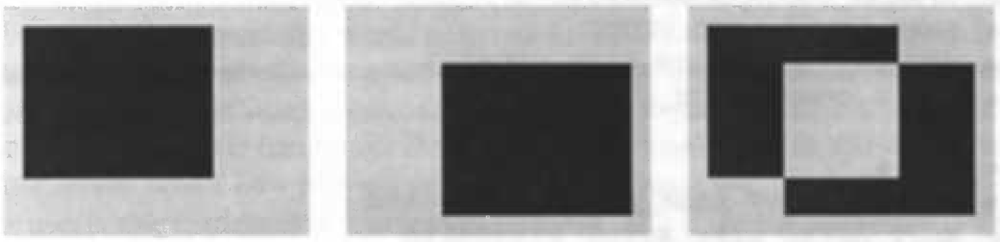


Figure 4.4: *Drawing using the XOR–function: (a) Window before drawing, (b) drawing shape, (c) result after drawing*

Now consider input #1 to be the color of one pixel in the window and input #2 to be the pixel color that has to be drawn in that window. Looking at the table, we can understand Figure 4.4. In Figure

4.4a the contents of the screen are displayed before drawing. In figure 4.4b, the shape that is going to be drawn is displayed and in Figure 4.4c the result of drawing the shape of Figure 4.4b in Figure 4.4a using the XOR–function is displayed.

As can be seen from Figure 4.4, drawing while using the XOR–function makes it possible to draw something which can be deleted very easily afterwards (by drawing the same shape again).

In `DrawElement()`, the graphics context is defined first. Graphics contexts are provided in X–Windows for defining graphical information which can be used for drawing. Drawing information provided by the graphics contexts can be the foreground color, the line width (for drawing lines, circles, etc.), the drawing function (e.g. the XOR–function), the line style (solid lines, dashed lines), etc. More information about the possibilities of graphics contexts can be found in [12].

The graphics contexts used in `DrawElement()` are defined in `ClassInitialize()`, using the constants defined in `visualize.c`:

```
/* Colors outside edit level */
#define GROUP_COLOR IxmColorDarkYellow
#define OPENED_GROUP_COLOR IxmColorDarkYellow
#define CONNECTION_COLOR IxmColorLightGray
#define EDIT_LEVEL_COLOR IxmColorOrange
#define SELECTED_COLOR IxmColorWhite
#define INPUT_COLOR IxmColorDarkGreen
#define HIDDEN_COLOR IxmColorDarkBlue
#define OUTPUT_COLOR IxmColorDarkRed

/* Colors inside edit level */
#define EDIT_GROUP_COLOR IxmColorYellow
#define EDIT_OPENED_GROUP_COLOR IxmColorYellow
#define EDIT_INPUT_COLOR IxmColorGreen
#define EDIT_HIDDEN_COLOR IxmColorBlue
#define EDIT_OUTPUT_COLOR IxmColorRed

/* Font */
#define ID_FONT IxmFontSHelvB12

/* Line width */
#define OPENED_GROUP_LINE_WIDTH 1
#define EDIT_LEVEL_LINE_WIDTH 3
#define NEURON_CONNECTION_LINE_WIDTH 1
#define GROUP_CONNECTION_LINE_WIDTH 3

/* Line style */
#define OPENED_GROUP_LINE_STYLE LineSolid
#define EDIT_LEVEL_LINE_STYLE LineSolid
```

Depending on the element type, neuron type, etc., the correct graphics context is determined and assigned to `use_gc`.

In the following sections we will discuss the visualization of the different elements that can exist in a network.

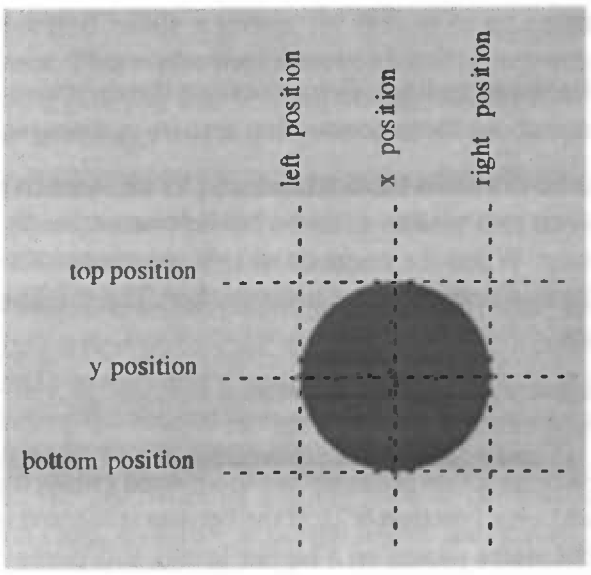


Figure 4.5: *Positioning of network elements*

4.2.1 Neurons

When the element type is a neuron, a triangle, rectangle or circle is drawn. For specifying the correct positions of these shapes, some routines have been defined in **`operations.h`** (section 6.7.1) for specifying the top, bottom, left and right positions of an element. They can be used for all element types and use the position of an element and its dimensions (the left position is defined to be the x position minus the half of the width of the element). Figure 4.5 gives an overview of these routines.

Since we also want to add neurons to existing groups, there must be a difference in visibility between the group in which neurons will be added and other groups. We do this by drawing the contents of a group brighter than the elements that are outside this group. Elements can be drawn with two functions: one using the XOR and one for normal drawing. Therefore there are twelve different graphics contexts for neurons ($3 \times 2 \times 2$, three neuron types, two functions and two levels of brightness).

Using the graphics context, defined in **`use_gc`**, we draw a triangle, square or rectangle (for inputs, hiddens and outputs). Inside these symbols more information about the neuron can be viewed. Possibilities of information to visualize within the neuron symbols are the neuron ID and the transfer function. A possibility would be to make the information user–specified. In this case the user is able to choose whether he wants to see the neuron ID, the transfer function or nothing at all. Since the network editor has not been completely merged in InterAct yet, there is no information about the transfer function or the neuron ID yet. Implementing this version of the network editor, every time a new element is created this element will get a unique ID. The only choice the user is able to make now is to see this ID or not. This ID can be visualized in the neuron. After merging the network in InterAct the "real" neuron ID within InterAct will be visualized in the neuron.

After drawing the neuron symbol (triangle, square or circle), **`GetWindowInfoShowID()`** is called to check whether ID's have to be shown in the network editor. If the user wants to see ID's, the ID of the neuron will be written in the center of the neuron.

4.2.2 Connections

Connections are being visualized as lines. For indicating the direction of a connection, we draw an arrow on this line. More about these connection arrows is discussed later in this section.

When a connection has to be drawn in `DrawElement()`, we want to look whether this connection is a connection between two visible neurons or the connection is a connection that is connected to an iconified group. When the connection is connected to an iconified group, we want to draw a line between the two cells that is a little thicker. The thickness of both connections is defined by these constants:

```
#define  NEURON_CONNECTION_LINE_WIDTH  1
#define  GROUP_CONNECTION_LINE_WIDTH  3
```

For determining whether a connection is connected to a closed group, the call `ClosedParent()` is used, defined in `operations` (section 6.7). If the neuron is located inside a closed parent (this can be the direct parent but also a parent on a higher level), this parent is returned. If the neuron is not located inside a closed parent (and therefore visible on the screen), the neuron itself is returned. So determining whether a thick line has to be drawn can be done by the following test:

```
if  (ClosedParent(GetConnectionFrom(e))  !=  GetConnectionFrom(e))  ||
     ClosedParent(GetConnectionTo(e))  !=  GetConnectionTo(e)))
```

If the result of this test is `True`, connection lines for groups are drawn (a different graphical context). Otherwise the normal graphical context is used. For drawing the connection, `DrawConnection()` is called.

DrawConnection

`DrawConnection()` is defined in `visualize` and has the following syntax:

```
void DrawConnection(w, gc, connection, type, info)
     Widget w;
     GC gc;
     Element *connection;
     ConnectionType type;   /* ARROW, LINE_AND_ARROW */
     WindowInfo *info;
```

The parameters define the widget to draw in, the graphics context to draw with, the connection to draw, the connection type and the window information. The connection type specifies whether `DrawConnection()` has to draw only the arrow or the line with the arrow. The advantage of just drawing an arrow will be explained later.

The first thing that is done in `DrawConnection()` is to look whether the connection has to be visualized. There are two reasons why a connection should not be visualized:

- if the user has defined that the connection is not visible;
- when a connection is connecting two cells inside a closed group.

To check whether the user wants to see a connection, function `ShowConnection()`, defined in `operations`, is called. This function examines the `ConnectionState` of the connection (and if necessary its parent, etc.) and returns `True` if the connection should be visible. The meaning of the `ConnectionState` has already been discussed in 4.1.1.

When the connection is located inside a group, the results of `ClosedParent()` of both connected cells must be the same. This is also the fact when a cell is connected to itself. Another property of a connection inside a group is that it is not visible. So the following test becomes `True` if we do not have to draw anything:

```
if  ((ClosedParent(GetConnectionFrom(e)) ==
       ClosedParent(GetConnectionTo(e))) &&
      !IsVisible(GetConnectionFrom(e)))
```

When a line has to be drawn (the `ConnectionType` is `LINE_AND_ARROW`), we check whether the connection is connecting a neuron to itself or whether this is a connection between different neurons. If the connection is connecting a neuron to itself, a circle is drawn above the neuron. If the connection is connecting the neuron to another neuron, a line is drawn using the calls `GetConnectionFromXPosition()`, `GetConnectionYPosition()`, etc. Also the position of the arrow is depending on the fact whether a connection is connecting a neuron to itself or to another neuron. In the first case, an arrow is drawn above the neuron. Arrows are drawn using `DrawArrow()`.

DrawArrow

The syntax of `DrawArrow()` is:

```
void DrawArrow(w, gc, x1, y1, x2, y2, info)
    Widget w;
    GC gc;
    int x1, y1, x2, y2;
    WindowInfo *info;
```

By just drawing lines to visualize connections, there will be no difference visible between the directions of the connections. Drawing indicators (arrows) on the connections makes it possible to see these differences. Drawing indicators at the middle of the connection will make the indicators of two multiple connected neurons interfere with eachother. Drawing indicators at the end of the connections will make more incoming connections at one neuron interfere with eachother. Drawing indicators between these positions (e.g. 2/3 of the total connection length) will result in a clear view.

Another problem drawing arrows is that we want the sizes of different arrows to be equal. When just calculating the positions of the handles of an arrow by using a sine or cosine, arrows on long lines will become larger than arrows on short lines. Using vector–algebra gives a solution to this problem.

By looking at figure 4.6, we want to determine the coordinates of R and S, using the coordinates of A and B, the factor k determining the location of the arrow (e.g. 2/3) and the distances e and f. Define $A = (x_1, y_1)$ and $B = (x_2 y_2)$. The equation vector connecting A and B will be:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \lambda \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{4.1}$$

This defines the coordinates of the front of the arrow (P) as:

$$P = k \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{4.2}$$

By determining the normalized vector between A and B and using the value of e, we can obtain the coordinates of Q:

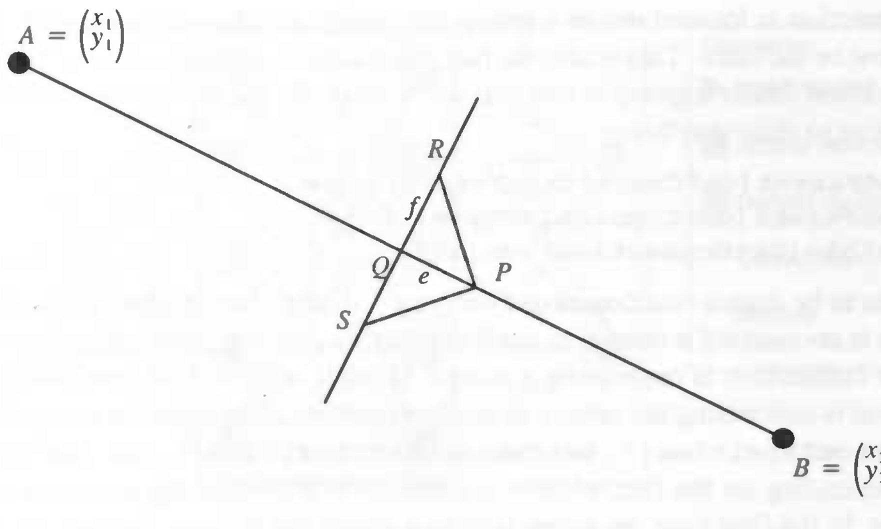


Figure 4.6: *Determining the position of an arrow symbol*

$$\binom{x}{y} = P + \lambda \frac{\binom{x_2 - x_1}{y_2 - y_1}}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{4.3}$$

$$Q = P - e \frac{\binom{x_2 - x_1}{y_2 - y_1}}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{4.4}$$

By determining the vector perpendicular to the vector in equation (4.3) and using the value of f, we can determine the coordinates of the left and right side of the indicator (R and S):

$$R = Q - f \frac{\binom{y_2 - y_1}{x_1 - x_2}}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{4.5}$$

$$S = Q + f \frac{\binom{y_2 - y_1}{x_1 - x_2}}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{4.6}$$

Defining $\delta = \frac{1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$, and substituting P in Q and Q in R and S, we obtain:

$$P = \begin{pmatrix}(1 - k)x_1 + kx_2 \\ (1 - k)y_1 + ky_2\end{pmatrix} \tag{4.7}$$

$$R = \begin{pmatrix}(1 - k + \delta e)x_1 + (k - \delta e)x_2 - \delta f y_2 + \delta f y_1 \\ (1 - k + \delta e)y_1 + (k - \delta e)y_2 - \delta f x_1 + \delta f x_2\end{pmatrix} \tag{4.8}$$

$$S = \begin{pmatrix}(1 - k + \delta e)x_1 + (k - \delta e)x_2 + \delta f y_2 - \delta f y_1 \\ (1 - k + \delta e)y_1 + (k - \delta e)y_2 + \delta f x_1 - \delta f x_2\end{pmatrix} \tag{4.9}$$

In `visualize`, we define constants for k, e and f (`FRONT_LOCATION`, `PERPEND_DIST` and `SIDE_DIST`). Using these constants, arrows can be drawn at the desired positions.

4.2.3 Modules

Groups are also drawn by `DrawElement()`. Groups can be visualized in different ways, depending on the properties of the group. Constants defining the visualization of groups are:

```
/* Colors outside edit level */
#define GROUP_COLOR IxmColorDarkYellow
#define OPENED_GROUP_COLOR IxmColorDarkYellow
#define EDIT_LEVEL_COLOR IxmColorOrange

/* Colors inside edit level */
#define EDIT_GROUP_COLOR IxmColorYellow
#define EDIT_OPENED_GROUP_COLOR IxmColorYellow

/* Line width */
#define OPENED_GROUP_LINE_WIDTH 1
#define EDIT_LEVEL_LINE_WIDTH 3

/* Line style */
#define OPENED_GROUP_LINE_STYLE LineSolid
#define EDIT_LEVEL_LINE_STYLE LineSolid
```

If the group is the group in which can be edited (the group is the current edit level), a solid orange line of 3 pixels will be drawn. Closed and opened groups that are not the current edit level will be drawn with using yellow, or dark yellow if they are not within the current edit level. The line style will be solid. At first, the diference between the edit level and other groups was visualized by using a dashed line around the other groups. Reason for not using this method is the calculation intensity for drawing dashed lines on a UNIX–machine.

Depending on the properties of a group, a different graphical context is used. Closed groups will be drawn as filled rectangles. Opened groups will be drawn as "open" rectangles. Since each group has a unique ID within the network, the ID's can be visualized too. Drawing ID's in an opened group will not provide a clear view of the network structure. When a group is closed, there is enough room to visualize the group ID in the filled rectangle. This is also handled in `visualize.c`.

4.2.4 Total network structure

In Figure 4.7 we can see an example of a network structure, visualized by the network editor. Six groups have been defined: one for the total network, one for the input layer, one for the output layer, one for the hidden layer and two for the separate hidden layers. As can be seen, the input and output layer are less bright drawn than the hidden layer. This is because the hidden layer is the current edit level. Another way to see this is the thickness of the border around the hidden layer. In the next section, we look at the process of defining and manipulating such a network.

4.3 Manipulation of the network structure

In this section we will discuss the manipulation of the network structure. As already mentioned in section 4.1.4, we have defined three modes to manipulate the network structure: edit mode, neuron mode and connection mode. The neuron mode is used to place neurons in the network structure. When selecting the connection mode, connections can be created between cells in the network. The edit mode is used to perform all other manipulations (moving, resizing, etc.).

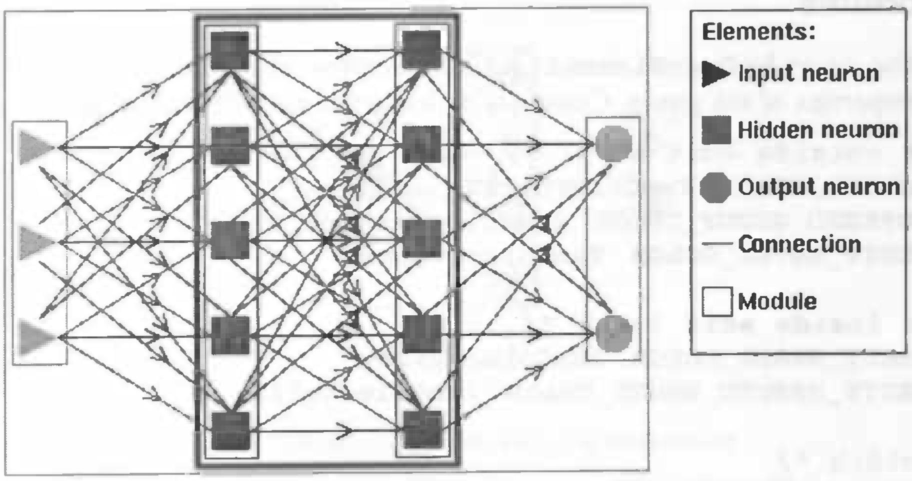


Figure 4.7: *Visualization of a network using the network editor*

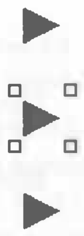


Figure 4.8: *Visualization of selecting objects*

Since most manipulation is done by mouse, the module **`drawing`** (see section 6.3) has been created to handle actions in the drawing area. In **`drawing.c`**, the functions **`DrawingButtonCallback()`** is created to handle **`ButtonPress`** and **`ButtonRelease`** events and **`DrawingMotionCallback()`** is created to handle the event that is generated by moving the mouse while holding a mouse button. More information about events can be found in [12].

Before discussing the implementation of the possible manipulations, we would like to mention the possibility to select objects in the network editor. After selecting objects, these objects can be manipulated (moving, copying, cutting, etc.). When new objects are created, they become selected. This makes it possible to directly change the properties of a newly created object or to cut an object that was created by mistake. Defining a new selected element is done by **`SetWindowInfoSelected(info, elt)`**. When the second argument is **`NULL`**, the list of selected elements will become empty. Visualizing selected items is done by drawing small "handles" at each corner of a selected item. Drawing these handles is done by calling **`DrawCorners()`**. Using handles to indicate the selection of an object is shown in Figure 4.8.

This section is divided in three subsections, describing the three modes that are available in the network editor.

4.3.1 Neuron mode

Description

The neuron mode makes it possible to add neurons to the current edit level. When the neuron mode is selected, pressing the mouse button at a position will imply the creation of a new neuron at that position in the current edit level. Adding a neuron to a group that is not big enough can be treated in two different ways: the group can be resized (maybe its parent too, etc.) to become big enough to contain the neuron or the neuron is not added to the group. Since there is also the possibility to resize a group, we choose the latter option.

Defining the neuron type is now possible by selecting one of the three possible neuron types (input neuron, hidden neuron or output neuron) in a menu in the network editor. Another possibility could be to offer three neuron modes, for each neuron type one.

Implementation

In `DrawingButtonCallback()`, `AddNeuron()` is called when the left mouse button was pressed while the user is working in neuron mode. This function is defined in `operations.c`. It checks whether the neuron is being added in the main group or within the borders of another group. If this is the case, a neuron is created using `CreateNeuron()` and added to the internals of the group using `AddGroupInternal()`. The new element is returned. In case of failure `NULL` is returned.

If the call to `AddNeuron()` was successful, the new element is drawn (without using the XOR–function) and `NewSelected()` is called. This function both defines the object to be selected and makes this selection visible by calling `SetWindowInfoSelected()` and `DrawSelected()`.

4.3.2 Connection mode

Description

The connection mode makes it possible to add connections to the network. Making connections is edit level independent. When the making of connections would be edit level dependent, connections from neurons inside one group to another group will not be possible.

Making a connection can be done by pressing the left mouse button on the neuron from which the connection has to be created and dragging the mouse to the neuron which receives the signal and releasing the mouse button. While dragging the mouse, a line will be drawn from the point where the mouse button was pressed to the current mouse position.

Making more connections can be done by performing the same routine. When for instance one wants to create a connection from an opened group, one has to press the mouse button inside that group (not on a neuron, in that case the connection will be created from that neuron). Another possible way of creating more connections would be to press the mouse pointer on the border of an opened group, but this requires from the user to look very closely where the mouse pointer is located. While dragging the mouse pointer, the object that would be considered the end of the connection when the mouse pointer is released, will be indicated by symbols at the corners of this object (see Figure 4.9). When releasing the mouse button at a position where no objects are present, no connections will be created. When releasing the mouse button on the same neuron the button was pressed implies the creation of a connection to the neuron itself.

When drawing connections, it is possible to make an assumption about the orientation which the user is using. When assuming the orientation of the network is "left–to–right", connections will be displayed as in figure 4.10. Figure 4.10a shows a clear view of the network. Only the the connection connecting the upper output neuron to the lower hidden neuron indicates that the system assumes a "left–to–right" orientation. When looking at figure 4.10b, one can see that designing

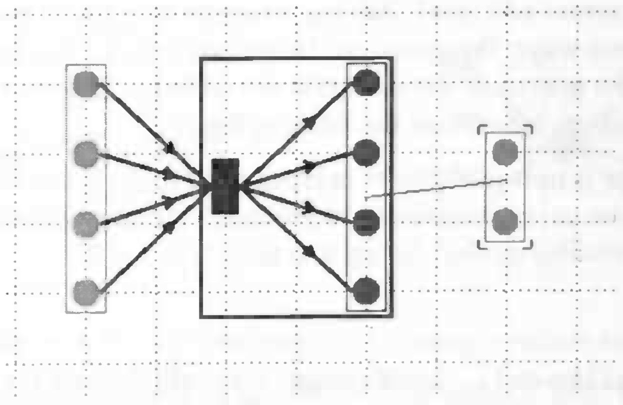


Figure 4.9: *Indicating the end of a connection*

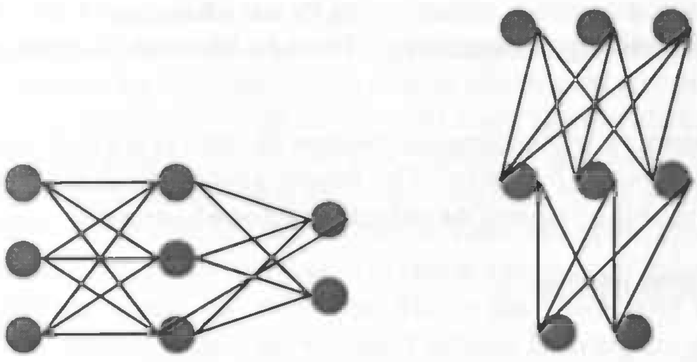


Figure 4.10: *(a) Left–to–right orientated connections while drawing with left–to–right orientation, (b) Left–to–right orientation while drawing with up–to–down orientation*

your system "up–to–down" while the interface is assuming a "left–to–right" orientation, doesn't result in a very clear view of the system.

Solving this problem, we have to look at the distance between the elements that have to be connected. When the horizontal distance is bigger than the vertical distance, connections will be drawn between the left and right side of the elements. The element being the most left of the two will be connected from the right side of the element to the left side of the other element. When the vertical distance is bigger than the horizontal distance, the connection will be drawn between the top and bottom side of the elements. The element lying above the other will be connected from the bottom side of the element to the top side of the other element.

Implementation

Implementing the creation of a connection, we need to look at `DrawingButtonCallback()`. In this function, the list of selected items will first be emptied. This is because we do not need any objects to be selected anymore. The variables `start_x` and `start_y` define the beginning of a connection and `last_x` and `last_y` define the current position of the mouse pointer. A line is drawn between these points (which are intialized to be equal) using the XOR–function. Also a variable `end_connection` is initialized to indicate the element that is currently at the position of the mouse pointer. `DrawBorders()` is implemented in `visualize.c`. It draws the corners around the element that is currently at the mouse pointer (see Figure 4.9).

Moving the mouse will imply a call to `DrawingMotionEvent()`. In this function, the old line is removed by drawing it again using the XOR–function and a new line is drawn to the new mouse positions. A call to `GetConnectionEnd()` (implemented in `operations.c`) checks whether a new object is at the mouse pointer. If this is the case, the old indicators are removed and new ones are drawn.

When the mouse button is being released, `DrawingButtonEvent()` is called again. The indicators and the line will be removed. A connection is created calling the function `Connect()` (implemented in `operations.c`, section 6.7.2). This function will call another function which recursively makes the specified connections. After a connection is created, `SetConnectionPositions()` is called, which will determine the direction independent positions of the connections using the method explained above. The new connections are added to the list of selected elements. After the connections have been created, the list of selected items is used to draw the new connections. `DrawCorners()` is used to draw the selection indicators.

4.3.3 Edit mode

The edit mode is probably the most important mode of the network editor. Using the edit mode, objects in the network can be selected so operations can be applied on them. Also the resizing and moving of elements are handled in the edit mode.

Description

Only objects in the current edit level can be selected. Selecting one object can simply be done by pressing the left mouse button on this object. Selecting more objects, while objects are already selected, is possible by pressing the left mouse button upon an object while holding the shift key. When pressing the mouse button on an empty part of the network editor, the list of selected objects will be emptied.

Another possibility of selecting objects is dragging the mouse around the objects. When the user presses the mouse not on an object, he can drag the mouse to another position, forming a rubber band. When releasing the left mouse button, all objects inside the rubber band will become selected. When the mouse is dragged while holding the shift key adds the objects inside the rubber band to the list of the already selected objects.

When dragging a rubber band, connections inside this rubber band are not selected. When connections will also be selected, it is very difficult to select objects in more layers that are connected to each other. In this case, the user has to select each layer separately. Since we choose not to select the connections inside a rubber band, we need to offer another possibility to select more connections. We will discuss the solution to this problem later in this section.

As already mentioned, selected objects are visualized by drawing small circles at the corners. In Figure 4.11 we can see an example of selecting objects by dragging a rubber band and the result. Note that the rubber band looks the same as the bounding boxes of the groups, but in the program it is drawn with a different color.

When pressing the left mouse button upon an object and holding the button while moving to another position will move the indicated object to this position. The release of the button will define the end position of the object. Moving more objects can be done by first selecting these objects and by moving one of the selected objects to a new position. The other objects will be moved too.

As explained in 3.2.2, moving connections is not very useful. Therefore, moving the mouse after pressing a connection can be considered as the start of a rubber band. We also have to be aware

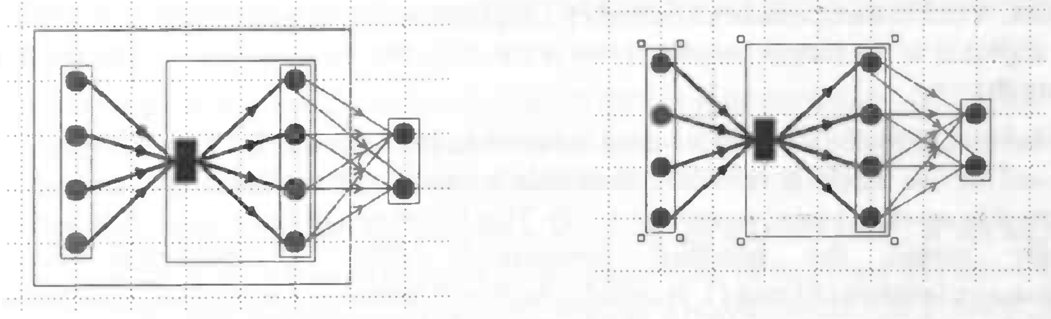


Figure 4.11: *Selecting more objects using a rubber band*

of the fact that a user will not be able to keep the mouse entirely still. Pressing an object while moving the mouse a little must not result in the movement of the object, but in the selection of the object.

Another mouse function is double click. A possibility to react on a double click would be to open a window to edit an element's properties. When double clicking on an opened group, we could use the double click to make this group the current edit level. Double clicking on a closed group could result in opening this group. Another way of "entering" and opening a group would be to create special buttons for these actions next to the drawing area. Since we do want to manipulate groups easily, we chose for the double click method to enter and open a group. Double clicking on other elements than groups could still be used to create a property window, but this would imply more than two different results on one action. This is the reason why we decided to simply handle a double click on other elements as two select actions. Leaving the group that is currently the edit level, can be done by pressing the mouse button outside the borders of the group. For this action separate buttons could be created too.

Taking these descriptions into account, we can generate a state diagram, as viewed in Figure 4.12. We will now describe the meaning of this state diagram, starting at state (0):

- Pressing the mouse button on a resize handle, will result in state (1). When bigger movements are noticed before releasing the button, we resize the group that the resize handles belong to. When only small movements are noticed before the release of the button, we check whether there is an element at the mouse pointer. If this is the case, this element will become selected. If there is no element at the mouse pointer, we deselect the selected items.
- If the user presses the mouse button at a position that is not in the current edit level or at a position where no element is drawn, we go to state (2). When only small movements are noticed before the release of the button, we go to state (7). If the button was pressed inside the edit level (or the main group is the current edit level), the selected objects become deselected. If the button was pressed outside the edit level in state (7), the edit level changes. The new edit level will be the parent of the old edit level. As soon as a bigger movement is noticed in state (2), we start to draw a rubberband until the button is released.
- Pressing the mouse button on a connection will result in state (3). When the mouse is moved too far, we start to draw a rubberband, just like before. When only small movements are measured, before the button is released, the connection is selected.
- If in state (0), the mouse button is pressed on a group, we move to state (4). If the mouse is not moved until the button release, the group is selected. After the selection of the

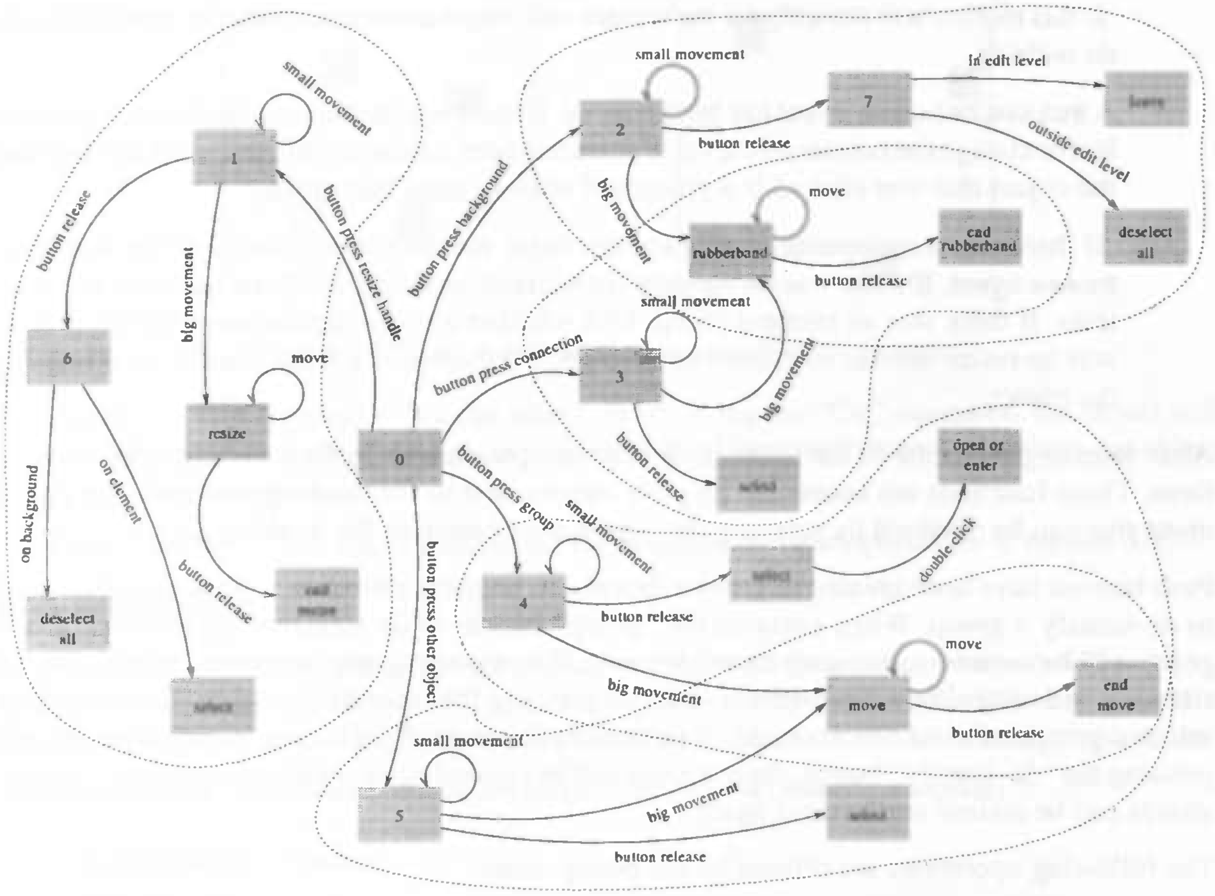


Figure 4.12: *State diagram of the edit mode*

group, a second mouse click can occur. In this case the group is opened or entered (depending on the current state of the group: closed or opened). When a bigger mouse movement is measured in state (4), the group is moved until the mouse button is released.

- If non of the above is true, state (5) is entered. In this case we are dealing with an ordinary object. If the mouse pointer is not moved too far, the object is selected. When a bigger movement is noticed, the object is moved to a new position.

Simplifying Figure 4.12, we can get the following rules:

- A `ButtonPress` event has been noticed. Check whether there is a resize handle. Remember the object that was pressed on or the object on which resize handle was pressed. Initialize a flag to indicate that no big movement has been measured yet.

- A `ButtonMotion` event has been noticed. If this motion is the first bigger movement, set a flag to indicate this. If there is no element at the ButtonPress, start drawing a rubber band. If there was an element, check whether there was a resize handle. If the button was pressed on a resize handle, start resizing the object. If there was no resize handle, start moving the object.

  If this motion was not the first bigger motion yet but there has been a bigger motion before, continue the already started resizing, moving or drawing of the rubber band.

If this motion was not a bigger movement and there has not been a bigger movement yet, do nothing.

- A `ButtonRelease` event has been noticed. If there was no movement, then select, deselect or change the current edit level. If there has been a mouse click before, check whether the object that was clicked is a group and open or enter this group.

  If there was a movement, check whether there was an element found at the `ButtonPress` event. If there was no element found, remove the rubber band and select its contents. If there was an element found, look whether a resize handle was pressed. If there was no resize handle, stop moving the object. If there was a resize handle, stop resizing the object.

After selecting elements in the network, it becomes possible to perform certain operations on them. These functions are accessible by push buttons next to the working area and by a popup menu that can be obtained by pressing the right mouse button in the working area.

Push buttons have been created to group elements, to ungroup elements, to iconify a group and to de–iconify a group. When pressing the "group" button while elements are selected, a new group will be created, containing these elements. When pushing the "ungroup" button, the old situation will return. When groups are selected, pressing the "iconify" button will visualize the selected groups as icons. their contents is not visible anymore. By selecting iconified groups and pressing the "de–iconify" button, these groups will be opened again. From this moment on, these groups can be resized and entered again.

The following operations are offered by the popup menu:

Properties Selecting "Properties" from the popup menu will result in a new popup window. In this window the properties of the selected item(s) are displayed and these properties can be changed. Since the network editor is not linked with the datastructure of InterAct, no properties can be changed yet.

Alignment This function is used to align elements. In section 3.2.2 the meaning of aligning elements has already been discussed. Alignment possiblities are: left, right, left–right–center, top, bottom and top–bottom–center. Figure 3.8 showed us the result of aligning three elements to their left–right–center position.

Adjust internals to group Performing this operation after selecting a group, will move the contents of the selected group to the borders of the group. Their relative distance will remain the same. An example of this function is given in Figure 4.13. This function makes it possible to use the total space within a group without resizing the group.

Adjust group to internals If internals have been removed from a group, this function can be used to resize the group as small as possible. This can also be done by resizing the group from the top–left and lower–right corner.

Select connections Since connections can not be selected using the rubber band method, this operation is offered to select more connections. By selecting an element, its incoming and outgoing connections can be selected. By selecting more elements, the connections that are connecting these cells (or its internals) can be selected.

Cut / copy / paste These operations have already been discussed in section 3.2.1.

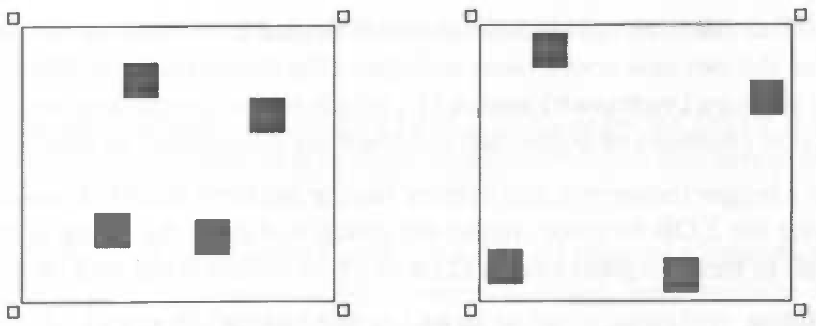


Figure 4.13: *Adjust internals to group: (a) before, (b) after*

Delete This function deletes an object. When using the "Cut" operation, the object will still remain in the buffer, so it can be pasted later. Using the "Delete" operation, the object will be removed.

Raise / lower We already discussed the meaning of these operations in section 3.2.2.

Implementation

For handling the mouse events in the edit mode, we first look at `DrawingButtonCallback()`, defined in `drawing.c` (section 6.3.2). In this function, we first look whether there was a resize handle at the button press using `GetResizeHandle()`. This function is defined in operations:

```
ResizeHandle  GetResizeHandle(info, x, y, element_return)

    WindowInfo  *info;
    int x, y;
    Element  **element_return;
```

where `Resizehandle` can be `NO_HANDLE`, `TOP_LEFT`, `BOTTOM_LEFT`, `TOP_RIGHT` or `BOTTOM_RIGHT`, depending on whether there is a resize handle and its position.

If there was a resize handle, `pressed_element` will become the element that is going to be resized. If there is no handle pressed, we will try to find an element in the current edit level which is not a connection, using `GetElementInEditLevel()`. If there exists a resize handle, the element that is going to be resized will become selected (this is done to deselect others). A flag `moved` is assigned to be `False`, indicating that there has been no bigger movements yet.

When `DrawingMotionCallback()` is called while the user is working in edit mode, we check whether this is the first bigger movement. If this is true, `moved` will be assigned `True`. If we are dragging a rubber band (`pressed_element` is `NULL`) using the shift key, we want to leave the selection corners. Otherwise they can be removed. If `pressed_element` is defined and there is no resize handle, we are moving an object. In this case we want to check whether this object was already selected and whether we have to move other objects too. After checking this, the object will be selected if it was not selected yet. If there is a resize handle, we call `StartResize()`, defined in `operations`, which initializes the resizing routine. If there was no element found at the button press, we start drawing a rubber band.

After checking whether this movement was the first bigger movement, we continue in `DrawingMotionCallback()`. If there has already been a bigger movement, we check whether there was an element defined in `DrawingButtonCallback()`. If an element has been defined and there is no resize handle, we move the elements to their new positions using `MoveAndDra-`

`wElements()`. This function removes the selected elements from their position using the XOR drawing function, defines new coordinates and draws the elements again. Moving is done by using the function `RecursiveMoveElement()`, which moves an element and its contents. Also the new connection positions of connected synapses are determined in this function.

If there has been a bigger movement and a resize handle has been detected, we remove the border of the group using the XOR function, resize the group and draw the group again. If no element was pressed at all in `DrawingButtonCallback()`, a rubber band will be drawn.

Releasing the button, will cause a call to `DrawingButtonCallback()`. If this is done in edit mode and there has not been a bigger movement, `Click()` is called. A second click of a double click has to be within the same area as the first click and it has to occur within short time. After calling `Click()`, we check whether there has already been a mouse click and whether this could be the second click of a double click. If this is the case, `DoubleClick()` is called. If this is not the case, variables are defined to remember the time and place of this click event, since this could be the first click of a double click.

If there has been a movement after the button press, we remove the rubberband if `pressed_element` is equal to `NULL` and `Drag()` is called.

Click() This function handles single mouse clicks. These clicks can occur when a user selects elemets, leaves the current edit level or wants to deselect all elelements. First, the element at the position of pressing the mouse button is retrieved. If there is no such element, we check whether there is a connection at that position using `GetConnectionAt-Position()`. If the mouse button was pressed while holding the Shift key, we add the element that was pressed to the list of selected objects using `AddWindowInfoSe-lected()`. If no Shift key was pressed, we look whether the current edit level is the main group or the position of the mouse press was within the current edit level. If this is the case, the element that was pressed becomes selected using `NewSelected()`. If no element was selected, `NewSelected()` will deselect all elements. If the mouse press was outside the current edit level, `LeaveGroup()` is called. This function is defined in `op-erations.c`. It changes the current edit level and redraws the working area.

DoubleClick() This function is only used for groups. It checks whether the group at the mouse position is opened or closed when the double click occured. If the group is opened, `EnterGroup()` (defined in `operations.c`) is called. this function changes the edit level to another group and redraws the drawing area. If the group was closed, it will be opened by using the function `DeIconifyCallback()`. This function is defined in `commands.c`.

Drag() Dragging the mouse can be done when moving an object, resizing a group or when a rubber band is created. If `pressed_element` is defined and there was no resize handle, the element has been moved so the drawing area has to be redrawn. If there was a resize handle, a group has been resized. We define the new position and dimension of the group and draw new selection corners around the group.

If `pressed_element` is `NULL`, we look at the corners of the rubber band. If the Shift key is used, the corners around the already selected items are removed (because they will be drawn again after defining the newly selected objects). If no Shift key was pressed, the list of selected objects is emptied. After this, all elements in the current edit level are being examined whether they are positioned within the corners of the rubber band. If this is the case, the element is added to the list of selected objects. After this the selection corners are drawn again.

All other manipulation functions, obtained by push buttons or by the popup menu, are defined in `commands.c`. The source code of this module is listed in 6.1.2. A discussion about those functions is listed below:

GroupCallback() If no elements are selected or all selected elements are connections, the function is quit. If there are other elements selected, we determine the position and dimension of the smallest group that could cover these elements. After this we create a new group in the current edit level, using `CreateGroup()`. All selected elements are removed from the current edit level and added to the new group. The new group is added to the current edit level. After this `PropagateSizes()` (defined in `operations.c`) is called to check whether the group exceeds the borders of its parent. If this is the case the size of the parent (and maybe its parent, etc.) is also changed in `PropagateSizes()`. If sizes have changed, the new object becomes selected and the working area has to be redrawn. If the sizes have not changed, we just draw the new element and call `NewSelected()` to make the new group the only selected object.

UnGroupCallback() First, we search the list of selected elements for groups that are selected. After searching the list, `groups` contains the selected groups. If no groups were selected, the function will end. The selection corners are removed and the list of selected objects becomes empty. After this, all group borders will be removed using the XOR, the contents of the group are defined to be the internal of the current edit level and they are added to the list of selected elements. The selected groups are removed from the current edit level and destroyed. After freeing the memory of `groups`, the new selection corners are drawn.

IconifyCallback() If there are no elements selected but the current edit level is not the main group, the current edit level is changed to the parent of the old edit level and the old edit level becomes selected. This makes it possible to iconify the current edit level. If the main group is the current edit level or elements have been selected, we check whether any groups have been selected. If not, we leave this function.

After this, we define the selected groups closed by calling `SetGroupOpened()` and define new internal connection positions (connections have to be drawn to the iconified group now). After this we call `PropagateSizes()` and redraw the drawing area.

DeIconifyCallback() All selected groups are opened using `SetGroupOpened()` and their internal connection positions are defined. After this, the sizes are propagated using `PropagateSizes()` and we redraw the drawing area.

Alignment Alignment functions have been created in commands.c to align the selected elements. For aligning elements, `AlignElements()` is called, which is defined in `operations.c`. `AlignElements()` determines the minimal return value `LeftPosition()` and `TopPosition()` and the maximal return value of `RightPosition()` and `BottomPosition()` of the selected elements to determine the new positions of the elements. After this the elements are moved using `RecursiveMoveElement()` (in `operations.c`), which moves the elements and their internals. `RecursiveMoveElements()` also defines the new connection positions.

AdjustInternalToGroup() This function moves the internals of a group so they occupy the total space that is available inside the group, without changing their relative distance. All selected groups that are opened and contain at least one element are used within this function. From their children, their center position and total width ad height is determined. The internals of that group are now moved to their new positions.

AdjustGroupToInternals() From each selected non–empty group that is opened we define the center position of its children and the total width and height they use. We use these values to define the new position and size of the group.

Select connections Selecting connections is done by the callback functions `SelectIncomingConnectionsCallback()`, `SelectOutgoingConnectionsCallback()` and `SelectConnectingConnectionsCallback()`. These functions call the function `SelectConnections()`, defined in `operations.c`. After this we redraw the drawing area. `SelectConnections()` calls `RecursiveSelectConnections()`, which fills the list of connections that have to become selected.

When `RecursiveSelectConnections()` is called, the list of selected elements is emptied, the connections that are in `connection_list` are added to the list of selected elements and the memory for the list of connections is freed.

If `RecursiveSelectConnetions()` is callled for a group, it calls itself to select the connections of the internals of the group. If we are dealing with a neuron, we continue looking at its connections. If the other side of the connection is connected to a neuron that is within the same selected element, we leave this function. If the connection is drawn to this neuron and we want to select the incoming connections, the connection is added to `connection_list`. If the connection is drawn from this neuron and we want to select the outgoing connections, the connection is added to the `connection_list`. If the other side of the connection is within a selected element and we want to select the connecting connections, we add this connection to the `connection_list`.

CutCallback() `CutCallback()` calls `Cut()`, defined in `operations.c`. This function removes only non–connections. First, the paste list of the window information is emptied. Then the function adds the selected elements to the paste–list and removes the element from the list of internals of the current edit level. After this it calls `_RecursiveCut()`, which recursively removes the connections that are connected to the element (or its children). After this, the function `SetWindowInfoPasteType()` is called to define the operation that has been applied on the elements in the paste list and the list of selected elements is emptied.

`_RecursiveCut()` removes all connections from the list of elements in the window info structure. Connections have to be removed from this list because selecting connections is edit level independant, so for selecting connections we look at this list instead of the list of internals of the current edit level.

CopyCallback() `CopyCallback()` calls `Copy()`, defined in `operations.c`. This function first clears the paste list of the window information. Then all selected elements that are no connection are added to the list and the paste type is defined by `SetWindowInfoPasteType()`.

PasteCallback() This function is called to paste objects that have been cut or have to be copied. `PasteCallback()` calls `Paste()`, defined in `operations.c`. In `Paste()`, the list of selected elements is emptied first. After this the positions and dimensions of the objects in the paste list that are no connections are obtained. This is used to determine the position at which the objects have to be pasted. The position of the popup menu is *(x, y)*, the minimal x–coordinate is *x1* and the minimal y–coordinate is *y1*. This means that we have to move the objects in the paste list over *(x – x1, y – y1)* to paste the objects at the current mouse pointer. After calling `_PasteAfterCut()` or `_PasteAfterCopy()`, depending on the type of paste list, `PropagateSizes()` is called and we redraw the window.

`_PasteAfterCut()` defines the new positions of the elements in the paste list by calling `RecursiveMoveElements()`. After this, the non–connections in the paste list are added to the current edit level and `_RecursivePasteAfterCut()` is called to connect the connections to the neurons that have been cut. For all connections in the paste list `SetConnectionPositions()` is called to determine the new positions of the connections. All other elements are added to the list of selected elements. Finally, the paste list is emptied.

`_PasteAfterCopy()` first initializes a list `list`. This is a list of neurons, connections and their copies. We discuss the usage of this list later. First all elements are copied using `_FirstRecursivePasteAfterCut()`. After this, `_SecondRecursivePasteAfterCut()` looks whether any neurons have been pasted that used to have connections connected to them and creates new connections. The pasted elements are added to the list of selected elements and `list` is freed.

`_FirstRecursivePasteAfterCopy()` looks at the element type to decide what has to be done. If the element is a neuron, a new neuron is created at the correct position and the neuron is added to its parent. The neuron and its copy are saved in the copy list `copy`. If the element is a group, a new group is created at the correct position and the group is added to its parent. After this `_FirstRecursivePasteAfterCopy()` is called to paste the internals of the group.

`_SecondRecursivePasteAfterCopy()` calls itself recursively until the parameter `e` is a neuron. Foreach connection of that neuron, we look whether the connection is already in the list of copies made. If no copy of this connection has been made yet, we continue. We create a new connection, connected to the copies of the neurons to which the connection used to be connected. After this, we define the connection positions and add the new connection to the list of copies already made.

DeleteElementCallback() This function calls `Delete()` to delete all selected objects. After this the list of selected objects is emptied and the network editor is redrawn. `Delete()` is also defined in `commands.c`. If the element that has to be deleted is a connection, the synapse is deleted from the list of synapses of the neurons that are connected to this synapse. Otherwise the element is removed from the list of internals of the parent of the element. If the element is a group, all its internals are deleted until the group does not contain any internals anymore. If the element is a neuron, all its connections are deleted until the neuron is not connected anymore.

Finally, the element is removed from the list of elements in the window information and the element is destroyed using `DestroyElement()`.

RaiseCallback() For all selected elements, we try to find their index within the list of elements in the window information. After this, we will define the selected element to be the last element in the list and move the rest of the list one position. This is done because drawing the network is done by first drawing element 0, then element 1, etc. Searching an element in the list is done by first looking at the last element, then the last but one element, etc., so putting the element at the back of the list will put the element at the back of the drawing list and at the starting of the search list. The same procedure is done with the internals of the parent of the selected items. Finally the network is drawn again.

LowerCallback() Where `RaiseCallback()` puts elements at the back of two lists, `LowerCallback()` puts elements at the starting of two list. After this the drawing area is redrawn.

Chapter 5 Evaluation

This chapter discusses the work done. The first section discusses how far we have achieved our goal. The second section discusses future work that can be done on the network editor and the third editor gives a conclusion.

5.1 Goal

First, we take a look at the goal presented in section 1.4:

Designing a graphical user interface for interactively creating and manipulating hierarchically based neural networks

The interface that has been written offers the possibility to create and manipulate a neural network which is hierarchically structured. To look more precisely how far we achieved our goal, we look at the demands, presented in 2.1:

1. It should be possible both to view the current network structure and to obtain characteristics of specific elements of this structure.
2. Network structures have to be manipulable. When manipulating neural networks it is very useful to manipulate more elements at the same time.

After creating a neural network using the network editor, the network can both be viewed and characteristics can be obtained. Since the network editor has not been linked to the internal datastructure of InterAct yet, not all properties can be obtained yet. Properties that are specific for a neural network (e.g. transfer function, ID's) can not be obtained yet.

The network structure is manipulable. Both structural and visualization operations can be used to change the network structure. Using the method of selecting more elements gives the opportunity to manipulate more elements at the same time.

5.2 Future work

As already explained above, the network editor has not been linked to the internal datastructure of InterAct yet. Creating a neuron in the network editor does not imply the creation of a neuron in InterAct. Furthermore, not all properties can be viewed or changed yet, since these properties are specific for neural networks. Examples of these properties are the ID's and the transfer functions of the neurons.

Using the Cut / copy / paste operations does not work correctly. Sometimes the program generates a runtime error. The cause of this error has not been found yet.

Some of the objects listed in 3.1 have not been implemented. The network editor does not offer the possibility to create fuzzy cells, algorithm cells or data cells yet.

5.3 Conclusion

5.3.1 Flexibility

Since the network editor has been written using the Abstract Data Type (ADT) principle, adding new elements to the network editor can be done very easily. The datastructure already offers the possibility to deal with fuzzy cells and algorithm cells.

5.3.2 Comparisation with other user interfaces

None of the graphical user interfaces for neural networks observed in 2.2 offer the possibility to create neural networks with a hierarchical structure. Interleaf 6 did offer the possibility to define groups and to change the edit level.

Also direct interaction using the mouse and manipulating more objects at the same time is another feature that was only offered by Interleaf 6. NeuroSolutions and NCS NEUframe were the only programs that offered a little mouse interaction, but not as extensive as the network editor that has been created now.

5.3.3 Usage

A group at the Rijksuniversiteit Groningen that is familiar with both neural networks and graphical user interfaces has tested the network editor thoroughly. The number of operations available to manipulate a hierachically structured neural network is sufficient. Building and manipulating a neural network can be done much faster using the network editor.

Using the operations available to manipulate the network, has been found intuitively. The visualization of the network status has been found very clear.

Chapter 6   Sources

6.1 Commands

6.1.1 commands.h

```
/*
* Header
*/
/*
* Log
*/


#ifndef COMMANDS_H
#define COMMANDS_H

#include <Ixm.h>


void GroupCallback(Widget, XtPointer, XtPointer);
void UnGroupCallback(Widget, XtPointer, XtPointer);
void IconifyCallback(Widget, XtPointer, XtPointer);
void DeIconifyCallback(Widget, XtPointer, XtPointer);
void DeleteCallback(Widget, XtPointer, XtPointer);
void LeftAlignCallback(Widget, XtPointer, XtPointer);
void RightAlignCallback(Widget, XtPointer, XtPointer);
void LeftRightCenterAlignCallback(Widget, XtPointer, XtPointer);
void TopAlignCallback(Widget, XtPointer, XtPointer);
void BottomAlignCallback(Widget, XtPointer, XtPointer);
void TopBottomCenterAlignCallback(Widget, XtPointer, XtPointer);
void SelectIncomingConnectionsCallback(Widget, XtPointer, XtPointer);
void SelectOutgoingConnectionsCallback(Widget, XtPointer, XtPointer);
void SelectConnectingConnectionsCallback(Widget, XtPointer, XtPointer);
void RaiseCallback(Widget, XtPointer, XtPointer);
void LowerCallback(Widget, XtPointer, XtPointer);
void AdjustInternalsCallback(Widget, XtPointer, XtPointer);
void AdjustGroupCallback(Widget, XtPointer, XtPointer);
void CutCallback(Widget, XtPointer, XtPointer);
void CopyCallback(Widget, XtPointer, XtPointer);
void PasteCallback(Widget, XtPointer, XtPointer);

#endif
```

6.1.2 commands.c

```
/*
* Header
```

```
*/
/*
* Log
*/


#include <macros.h>
/*#include <malloc.h>*/
#include <stdio.h>
#include "commands.h"
#include "interface.h"
#include "operations.h"
#include "visualize.h"


/* Utility for determining whether a number is positive / negative
   Returns 1 if positive, -1 if negative and 0 if zero */
#define signof(x) ((x) ? (((x) > 0) ? 1 : -1) : 0)
extern void SetDefaultMode();


/* Callback to group already selected elements.
   Only elements which are no connections can be grouped
   */
void GroupCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *edit = GetWindowInfoEditLevel(info);
   Element *new;
   Position x, y;
   Dimension width, height;
   int i;


   /* Leave if no elements selected */
   if (!GetWindowInfoNumSelected(info))
      return;

   /* Search for any non-connection */
   for (i = 0; (i < GetWindowInfoNumSelected(info)) &&
        (GetElementType(GetWindowInfoSelected(info, i)) == CONNECTION_TYPE);
    i++) ;
   if (i == GetWindowInfoNumSelected(info))
      return;  /* Only connections were selected */

   /* Determine the middle (x,y) and total width / height of selected items */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      if (GetElementType(GetWindowInfoSelected(info, i)) != CONNECTION_TYPE)
         ProcessPositionAndDimension(GetWindowInfoSelected(info, i));
   GetPositionAndDimension(&x, &y, &width, &height);

   /* Create new group, add selected elements and make the new group
      a child of the current editlevel */
   new = CreateGroup(edit, x, y, width + GROUP_PADDING * 2,
                     height + GROUP_PADDING * 2, True, True, PARENT);
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      if (GetElementType(GetWindowInfoSelected(info, i)) != CONNECTION_TYPE) {
         RemoveGroupInternal(edit, GetWindowInfoSelected(info, i));
         AddGroupInternal(new, GetWindowInfoSelected(info, i));
      }
   AddGroupInternal(edit, new);

   /* If the size of the editlevel has changed, the area has to
      be redrawn. If not, only the added element has to be redrawn */
   if (PropagateSizes(edit)) {
      SetWindowInfoSelected(info, new);
      InterfaceReDraw(info);
   }
   else {
      DrawElement(new, True);
      NewSelected(info, new);
   }

   /* Switch to the default mode */
   SetDefaultMode(info);
}


/* Callback to ungroup already selected elements.
   Only elements which are opened groups can be ungrouped
   */
void UnGroupCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *edit = GetWindowInfoEditLevel(info);
   Element **groups, *e;
   int i, j, num_groups;

   /* Initialize list of groups to be ungrouped */
   groups = NULL;
   num_groups = 0;

   /* Search for any opened group */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      if ((GetElementType(GetWindowInfoSelected(info, i)) == GROUP_TYPE) &&
          GetGroupOpened(GetWindowInfoSelected(info, i))) {
         groups = (Element **) realloc(groups,
                                       (num_groups + 1) * sizeof(Element*));
         groups[num_groups] = GetWindowInfoSelected(info, i);
         num_groups++;
      }

   /* Check whether no opened groups were selected at all */
   if (!num_groups)
      return;

   /* No elements selected anymore (or yet..) */
   DrawCorners(info);
   SetWindowInfoSelected(info, NULL);

   /* Make childs of each group childs of current editlevel and destroy
      the group */
   for (i = 0; i < num_groups; i++) {
      DrawElement(groups[i], True);
      for (j = 0; j < GetGroupNumInternals(groups[i]); j++) {
         e = GetGroupInternal(groups[i], j);
         SetElementParent(e, edit);
         AddGroupInternal(edit, e);
         AddWindowInfoSelected(info, e);
      }
      RemoveGroupInternal(edit, groups[i]);
      RemoveWindowInfoElement(info, groups[i]);
      DestroyElement(groups[i]);
   }

   /* Free the list of groups */
   free(groups);
   /* Draw the selected elements and set the default mode */
   DrawSelected(info);
   SetDefaultMode(info);
}


/* Callback to iconify selected opened groups. If nothing is selected
   and the current editlevel is an opened group, this group will be
   iconified */
void IconifyCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *edit = GetWindowInfoEditLevel(info);
   Element *e;
```

```
    int i;


    /* Check whether no group was selected */
    if (!GetWindowInfoNumSelected(info) &&
        (edit != GetWindowInfoMainGroup(info))) {
       SetWindowInfoEditLevel(info, GetGroupParent(edit));
       SetWindowInfoSelected(info, edit);
    }
    else {
       /* Check whether there is an opened group selected */
       for (i = 0; (i < GetWindowInfoNumSelected(info)) &&
          (GetElementType(GetWindowInfoSelected(info, i)) != GROUP_TYPE) &&
          !GetGroupOpened(GetWindowInfoSelected(info, i)); i++) ;
       if (i == GetWindowInfoNumSelected(info))
          return;  /* No opened groups found */
    }

    /* Iconify each opened group and rearrange the positions of the
       connections to that group */
    for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
       e = GetWindowInfoSelected(info, i);
       if ((GetElementType(e) == GROUP_TYPE) && GetGroupOpened(e)) {
          SetGroupOpened(e, False);
          RecursiveSetConnectionPositions(e);
       }
    }

    /* Resize editlevel if necessary, redraw and set the default mode */
    PropagateSizes(edit);
    InterfaceReDraw(info);
    SetDefaultMode(info);
}


/* Callback to de-iconify selected elements.
   Only closed groups can be de-iconified
   */
void DeIconifyCallback(Widget w, XtPointer client, XtPointer call)
{
    WindowInfo *info = (WindowInfo *) client;
    Element *e;
    int i;


    /* Open all selected opened groups. Set the connectionpositions of the
       newly opened groups */
    for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
       e = GetWindowInfoSelected(info, i);
       if ((GetElementType(e) == GROUP_TYPE ) && !GetGroupOpened(e)) {
          SetGroupOpened(e, True);
          RecursiveSetConnectionPositions(e);
       }
    }

    /* Propagate the sizes of the editlevel, redraw the the network
       and select the default mode */
    PropagateSizes(GetWindowInfoEditLevel(info));
    InterfaceReDraw(info);
    SetDefaultMode(info);
}


/* Function to delete an element recursively. If the element is a group,
   the internals will be deleted first. If the element is a neuron, the
   connections will be deleted first.
   */
static void Delete(WindowInfo *info, Element *e)
{
    if (GetElementType(e) == CONNECTION_TYPE) {
       RemoveNeuronConnection(GetConnectionFrom(e), e);
```

```
      RemoveNeuronConnection(GetConnectionTo(e), e);
   }
   else {
      RemoveGroupInternal(GetElementParent(e), e);
      if (GetElementType(e) == GROUP_TYPE)
         while (GetGroupNumInternals(e))
            Delete(info, GetGroupInternal(e, 0));
      else if (GetElementType(e) == NEURON_TYPE)
         while (GetNeuronNumConnections(e))
            Delete(info, GetNeuronConnection(e, 0));
   }
   RemoveWindowInfoElement(info, e);
   DestroyElement(e);
}


/* Callback to delete selected elements
   */
void DeleteCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   int i;


   /* Delete the selected elements recursively */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      Delete(info, GetWindowInfoSelected(info, i));

   /* Nothing is selected anymore, the network has to be redrawn and
      the default mode can be set */
   SetWindowInfoSelected(info, NULL);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the left
   */
void LeftAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, LEFT);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the right
   */
void RightAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, RIGHT);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the horizontal center
   */
void LeftRightCenterAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, LEFT_RIGHT_CENTER);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the top
   */
void TopAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, TOP);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the bottom
   */
void BottomAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, BOTTOM);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback the align selected elements to the vertical center
   */
void TopBottomCenterAlignCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   AlignElements(info, TOP_BOTTOM_CENTER);
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback to select all the incoming connections of
   the selected elements
   */
void SelectIncomingConnectionsCallback(Widget w, XtPointer client,
                                       XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   SelectConnections(info, INCOMING);
   InterfaceReDraw(info);
}


/* Callback to selected all the outgoing connections of
   the selected elements
   */
void SelectOutgoingConnectionsCallback(Widget w, XtPointer client,
                                       XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   SelectConnections(info, OUTGOING);
   InterfaceReDraw(info);
}
```

```
/* Callback to select all the connections that connect
   the selected elements
   */
void SelectConnectingConnectionsCallback(Widget w, XtPointer client,
                                         XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;


   SelectConnections(info, CONNECTING);
   InterfaceReDraw(info);
}


/* Callback to raise the selected elements (like if they were added last)
   */
void RaiseCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *edit = GetWindowInfoEditLevel(info);  /* parent of selected items */
   Element *e;
   int i, j;


   for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
      e = GetWindowInfoSelected(info, i);

      /* Find the selected element in the element list */
      for (j = 0; GetWindowInfoElement(info, j) != e; j++) ;
      /* Move the rest of the elements one place and put the selected element
         at the back at the list (elements at the back of the list are
         added last) */
      while (j < GetWindowInfoNumElements(info) - 1) {
         SetWindowInfoElement(info, j, GetWindowInfoElement(info, j + 1));
         j++;
      }
      SetWindowInfoElement(info, GetWindowInfoNumElements(info) - 1, e);

      /* Now find the element in the parent's childs and put element
         at back of this list */
      for (j = 0; GetGroupInternal(edit, j) != e; j++) ;
      while (j < GetGroupNumInternals(edit) - 1) {
         SetGroupInternal(edit, j, GetGroupInternal(edit, j + 1));
         j++;
      }
      SetGroupInternal(edit, GetGroupNumInternals(edit) - 1, e);
   }

   /* Redraw and set default mode */
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback to lower the selected elements (like if they were added first)
   */
void LowerCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *edit = GetWindowInfoEditLevel(info);  /* parent of selected items */
   Element *e;
   int i, j;


   for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
      e = GetWindowInfoSelected(info, i);

      /* Find the selected element in the element list */
      for (j = 0; GetWindowInfoElement(info, j) != e; j++) ;

      /* Move the rest of the elements one place and put the selected element
```

```
         at the front of the list (elements at the front of the list are
         added first) */
      while (j > 0) {
         SetWindowInfoElement(info, j, GetWindowInfoElement(info, j - 1));
         j--;
      }
      SetWindowInfoElement(info, 0, e);

      /* Now find the element in the parent's childs and put element
         at front of this list */
      for (j = 0; GetGroupInternal(edit, j) != e; j++) ;
      while (j > 0) {
         SetGroupInternal(edit, j, GetGroupInternal(edit, j - 1));
         j--;
      }
      SetGroupInternal(edit, 0, e);
   }
   /* Redraw and set default mode */
   InterfaceReDraw(info);
   SetDefaultMode(info);
}


/* Callback to move the internals of a selected group, while not changing
   the relative horizontal / vertical distance, so that they occupy
   the total width / height of the group
   */
void AdjustInternalsCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *e, *child;
   Position x, y;
   Dimension width, height, group_width, group_height;
   int i, j;


   for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
      e = GetWindowInfoSelected(info, i);

      /* Only move elements if the selected item is an opened group with
         at least one internal */
      if ((GetElementType(e) == GROUP_TYPE) && GetGroupOpened(e) &&
          GetGroupNumInternals(e)) {

         /* Calculate the middle (x,y) and the total width and height
            of the internals */
         for (j = 0; j < GetGroupNumInternals(e); j++)
            ProcessPositionAndDimension(GetGroupInternal(e, j));
         GetPositionAndDimension(&x, &y, &width, &height);

         /* The total width the internals may occupy is equal to the
            groupsize minus the space used for padding */
         group_width = ExternalWidth(e) - 2 * GROUP_PADDING;
         group_height = ExternalHeight(e) - 2 * GROUP_PADDING;

         /* Move the internals */
         for (j = 0; j < GetGroupNumInternals(e); j++) {
            child = GetGroupInternal(e, j);
            RecursiveMoveElement(child,
                              ((GetElementXPosition(child) - x +
                                signof(GetElementXPosition(child) - x) *
                                ExternalWidth(child) / 2) * group_width) /
                              width + GetElementXPosition(e) -
                              GetElementXPosition(child) -
                              signof(GetElementXPosition(child) - x) *
                              (ExternalWidth(child) / 2),
                              ((GetElementYPosition(child) - y +
                                signof(GetElementYPosition(child) - y) *
                                ExternalHeight(child) / 2) * group_height) /
                              height + GetElementYPosition(e) -
                              GetElementYPosition(child) -
```

```
                                       signof(GetElementYPosition(child) - y) *
                                       (ExternalHeight(child) / 2));
            }
         }
      }
      /* Redraw and set default mode */
      InterfaceReDraw(info);
      SetDefaultMode(info);
   }


   /* Callback to resize selected groups so they use a minimum width / height,
      while still covering the internals
      */
   void AdjustGroupCallback(Widget w, XtPointer client, XtPointer call)
   {
      WindowInfo *info = (WindowInfo *) client;
      Element *e;
      Position x, y;
      Dimension width, height;
      int i, j;


      for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
         e = GetWindowInfoSelected(info, i);

         /* Only proceed if the selected item is an opened group with
            at least one internal */
         if ((GetElementType(e) == GROUP_TYPE) && GetGroupOpened(e) &&
             GetGroupNumInternals(e)) {

            /* Calculate the middle (x,y) and the total width and height
               of the internals */
            for (j = 0; j < GetGroupNumInternals(e); j++)
               ProcessPositionAndDimension(GetGroupInternal(e, j));
            GetPositionAndDimension(&x, &y, &width, &height);

            /* Define the new position and size of the group. The size is equal
               to the width of the internals plus the group padding */
            SetElementXPosition(e, x);
            SetElementYPosition(e, y);
            SetElementWidth(e, width + 2 * GROUP_PADDING);
            SetElementHeight(e, height + 2 * GROUP_PADDING);
         }
      }
      /* Redraw and set default mode */
      InterfaceReDraw(info);
      SetDefaultMode(info);
   }


   /* Callback for cutting selected elements
      */
   void CutCallback(Widget w, XtPointer client, XtPointer call)
   {
      Cut((WindowInfo *) client);
   }


   /* Callback for copying selected elements
      */
   void CopyCallback(Widget w, XtPointer client, XtPointer call)
   {
      Copy((WindowInfo *) client);
   }


   /* Callback for pasting selected elements
      */
   void PasteCallback(Widget w, XtPointer client, XtPointer call)
   {
```

```
   Paste((WindowInfo *) client);

   /* Set the default mode */
   SetDefaultMode((WindowInfo *) client);
}
```

6.2 Constants

6.2.1 constants.h

```
#define GROUP_WIDTH 20                               /* width of iconified group */
#define GROUP_HEIGHT 40                             /* height of iconified group */
#define NEURON_WIDTH 20                               /* width/height of neuron */
#define GROUP_PADDING 4           /* padding between group-border and internals */
#define SELECTED_PADDING 5   /* space between element and "selection-circles" */
#define SELECTED_WIDTH 6                         /* width of "selection-circles" */
#define CONNECTION_DISTANCE 4  /* distance between mouse-click and connection */
```

6.3 Drawing

6.3.1 drawing.h

```
/*
* Header
*/
/*
* Log
*/


#ifndef DRAWING_H
#define DRAWING_H

#include <Ixm.h>

void DrawingExposeCallback(Widget, XtPointer, XtPointer);
void DrawingButtonCallback(Widget, XtPointer, XEvent*, Boolean*);
void DrawingMotionCallback(Widget, XtPointer, XEvent*, Boolean*);


#endif
```

6.3.2 drawing.c

```
/*
* Header
*/
/*
* Log
*/


/*
#include <report.h>
*/
#include <macros.h>
#include <stdio.h> /* TMP */
#include "drawing.h"
#include "interface.h"
#include "operations.h"
#include "visualize.h"
#include "commands.h"


#define MAX_CLICK_SPACE 10     /* Press and release of click < 10 pixels apart */
#define MAX_DBL_CLICK_SPACE 10                   /* Doubleclick < 10 pixels apart */
#define MOVE_SPACE 10            /* Space to drag button before moving/dragging */
#define DrawRubberband()\
   DrawArea(GetWindowInfoDrawingArea(info), min(start_x, last_x), \
   min(start_y, last_y), max(start_x, last_x), max(start_y, last_y))
#define DrawResizedGroup()\
   DrawGroup(GetWindowInfoDrawingArea(info), start_x, start_y, last_x, last_y);

#ifdef HANS
extern double IxmConvert(Widget, int, int);
#endif
extern void SetDefaultMode();
static int start_x = -1, start_y, last_x, last_y;       /* Connection drawing */
static Boolean moved;             /* Did the mouse move too much after press? */
static Element *pressed_element;                /* Element clicking mouse button */
static Element *end_connection;                         /* End of connection */
static Boolean pressed_is_selected;   /* Pressed element was already selected */
static ResizeHandle resize_handle;    /* Position of resizehandle / no handle */


static void MoveAndDrawElements(WindowInfo *info, int x, int y)
{
   Element *edit, *e;
   int dx, dy, i;


   dx = x - last_x;
   dy = y - last_y;

   /* Check whether one of the selected items is crossing a group border */
   edit = GetWindowInfoEditLevel(info);
   if (edit != GetWindowInfoMainGroup(info))
      for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
         e = GetWindowInfoSelected(info, i);
         dx = max(dx, LeftPosition(edit) + GROUP_PADDING - LeftPosition(e));
         dx = min(dx, RightPosition(edit) - GROUP_PADDING - RightPosition(e));
         dy = max(dy, TopPosition(edit) + GROUP_PADDING - TopPosition(e));
         dy = min(dy,
                  BottomPosition(edit) - GROUP_PADDING - BottomPosition(e));
      }

   if (dx || dy) {
      for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
         RecursiveDrawElement(GetWindowInfoSelected(info, i), True);
         RecursiveMoveElement(GetWindowInfoSelected(info, i), dx, dy);
         RecursiveDrawElement(GetWindowInfoSelected(info, i), True);
      }
      last_x = last_x + dx;
      last_y = last_y + dy;
   }
}


static void Click(XButtonEvent event, WindowInfo *info)
{
   Element *e, *edit;
   int x = event.x, y = event.y;


#ifdef HANS
   x = (int) IxmConvert (GetWindowInfoDrawingArea (info), 3, x);
   y = (int) IxmConvert (GetWindowInfoDrawingArea (info), 0, y);
#endif

   /* Only proceed if press and release are not too far away and in edit-mode */
   if ((abs(start_x - x) > MAX_CLICK_SPACE) ||
       (abs(start_y - y) > MAX_CLICK_SPACE) ||
       (GetWindowInfoMode(info) != EDIT_MODE))
      return;

   /* Find the element in which was clicked */
   e = GetElementInEditLevel(info, start_x, start_y, ALL_MASK, NO_MASK);
   edit = GetWindowInfoEditLevel(info);

   /* Maybe a connection is selected */
   if (!e)
      e = GetConnectionAtPosition(info, start_x, start_y);

   /* Maybe a connection */
   fprintf(stderr, "Selected ID=%d, EditLevel=%d\n",
           GetElementID(e), GetElementID(edit));

   if (event.state & ShiftMask) {
      DrawCorners(info);
      AddWindowInfoSelected(info, e);
      DrawSelected(info);
   }
   else
      if ((edit == GetWindowInfoMainGroup(info)) ||
          InElement(start_x, start_y, edit))
         NewSelected(info, e);
      else
         LeaveGroup(info, edit);
}


static void DoubleClick(XButtonEvent event, WindowInfo *info)
{
   Element *e;


   /* Only possible in edit-mode, with only one element selected
      with element type GROUP_TYPE */
   if ((GetWindowInfoMode(info) != EDIT_MODE) ||
       (GetWindowInfoNumSelected(info) != 1) ||
       (GetElementType((e = GetWindowInfoSelected(info, 0))) != GROUP_TYPE))
      return;

   if (!GetGroupOpened(e)) {
      DeIconifyCallback(NULL, (XtPointer) info, NULL);
      /* Redraw element because the element is selected and will blink */
      DrawElement(e, True);
   }
   else
      EnterGroup(info, e);
}


static void Drag(XButtonEvent event, WindowInfo *info)
{
   Element *e;
   int i;
   int min_x, max_x, min_y, max_y;


   fprintf(stderr, "buttonstate: %d\n", event.state);
   if (pressed_element) {
      if (resize_handle == NO_HANDLE)
         InterfaceReDraw(info);
      else {
         SetElementXPosition(pressed_element, (start_x + last_x) / 2);
         SetElementYPosition(pressed_element, (start_y + last_y) / 2);
         SetElementWidth(pressed_element, last_x - start_x);
         SetElementHeight(pressed_element, last_y - start_y);
         DrawCorners(info);
      }
   }
   else if (!(event.state & ~(Button1Mask | ShiftMask))) {
      min_x = min(start_x, last_x);
      max_x = max(start_x, last_x);
      min_y = min(start_y, last_y);
      max_y = max(start_y, last_y);
      if (event.state & ShiftMask)
         DrawSelected(info);
      else
         SetWindowInfoSelected(info, NULL);
```

```
        for (i = 0; i < GetGroupNumInternals(GetWindowInfoEditLevel(info)); i++) {
           e = GetGroupInternal(GetWindowInfoEditLevel(info), i);
           if ((min_x <= LeftPosition(e)) && (max_x >= RightPosition(e)) &&
               (min_y <= TopPosition(e)) && (max_y >= BottomPosition(e)))
              AddWindowInfoSelected(info, e);
        }
        DrawSelected(info);
     }
  }


void DrawingButtonCallback(Widget w, XtPointer client, XEvent *event,
                           Boolean *flag)
{
   WindowInfo *info = (WindowInfo *) client;
   Element *new;
   Position x, y;
   static Boolean clicked = False;
   static int x_click, y_click;
   static Time time_click;
   static Time max_doubleclick_time = 0;


   if (!max_doubleclick_time)
      max_doubleclick_time = XtGetMultiClickTime(Ixm_display);

   x = event->xbutton.x;
   y = event->xbutton.y;

#ifdef HANS
   x = (Position) IxmConvert (GetWindowInfoDrawingArea (info), 3, x);
   y = (Position) IxmConvert (GetWindowInfoDrawingArea (info), 0, y);
#endif

   switch (event->type) {
      case ButtonPress:
         if (event->xbutton.button == Button1) {
            if ((GetWindowInfoMode(info) == NEURON_MODE) &&
                (new = AddNeuron(info, x, y))) {
               DrawElement(new, False);
               NewSelected(info, new);
               SetDefaultMode(info);
            }
            else {
               start_x = last_x = x;
               start_y = last_y = y;
               if ((GetWindowInfoMode(info) == CONNECTION_MODE) &&
                   (pressed_element = GetConnectionEnd(info, x, y))) {
                  NewSelected(info, NULL);
                  DrawLine(GetWindowInfoDrawingArea(info), start_x, start_y,
                           last_x, last_y);
                  end_connection = pressed_element;
                  DrawBorders(end_connection);
               }
               if (GetWindowInfoMode(info) == EDIT_MODE) {
                  resize_handle = GetResizeHandle(info, x, y, &pressed_element);
                  if (resize_handle == NO_HANDLE) {
                     pressed_element = GetElementInEditLevel(info, x, y, ALL_MASK,
                                                             CONNECTION_MASK);
                     if (pressed_element) {
                        last_x = GetElementXPosition(pressed_element);
                        last_y = GetElementYPosition(pressed_element);
                     }
                  }
                  else {
                     SetWindowInfoSelected(info, pressed_element);
                     fprintf(stderr, "handle found: %d, eltID=%d\n",
                             resize_handle, GetElementID(pressed_element));
                  }
               }
               moved = False;
               }
            }
            break;
         case ButtonRelease:
            if (start_x != -1) {
               if (GetWindowInfoMode(info) == CONNECTION_MODE) {
                  DrawBorders(end_connection);
                  DrawLine(GetWindowInfoDrawingArea(info), start_x, start_y,
                        last_x, last_y);
                  start_x = -1;
                  if (end_connection) {
                     Connect(info, pressed_element, end_connection);
                     SetDefaultMode(info);
                  }
               }
               else if (!moved) {
                  Click(event->xbutton, (WindowInfo *) client);
                  if (clicked &&
                      (event->xbutton.time - time_click <= max_doubleclick_time) &&
                      (abs(x - x_click) <= MAX_DBL_CLICK_SPACE) &&
                      (abs(y - y_click) <= MAX_DBL_CLICK_SPACE)) {
                     DoubleClick(event->xbutton, (WindowInfo *) client);
                     clicked = False;
                  }
                  else {
                     time_click = event->xbutton.time;
                     x_click = x;
                     y_click = y;
                     clicked = True;
                  }
               }
               else {
                  if (!pressed_element)
                     DrawRubberband();
                  Drag(event->xbutton, (WindowInfo *) client);
               }
            }
            start_x = -1;
            break;
   }
}


void DrawingMotionCallback(Widget w, XtPointer client, XEvent *event,
                           Boolean *flag)
{
   WindowInfo *info = (WindowInfo *) client;
   int x, y, i;


   if (!(event->xmotion.state & Button1Mask))
      return;

   x = event->xmotion.x;
   y = event->xmotion.y;

#ifdef HANS
   x = (int) IxmConvert (GetWindowInfoDrawingArea (info), 3, x);
   y = (int) IxmConvert (GetWindowInfoDrawingArea (info), 0, y);
#endif

   if ((GetWindowInfoMode(info) == CONNECTION_MODE) && (start_x != -1) &&
       pressed_element) {
      DrawLine(GetWindowInfoDrawingArea(info), start_x, start_y,
            last_x, last_y);
      last_x = x;
      last_y = y;
      DrawLine(GetWindowInfoDrawingArea(info), start_x, start_y,
            last_x, last_y);
      if (GetConnectionEnd(info, x, y) != end_connection) {
         DrawBorders(end_connection);
         end_connection = GetConnectionEnd(info, x, y);
         DrawBorders(end_connection);
      }
   }
   if ((GetWindowInfoMode(info) == EDIT_MODE) && (start_x != -1)) {
      if (!moved && ((abs(x - start_x) > MOVE_SPACE) ||
                     (abs(y - start_y) > MOVE_SPACE))) {
         moved = True;
         /* Only leave corners if adding new selected items */
         if (!((event->xmotion.state & ShiftMask) && !pressed_element))
            DrawCorners(info);
         if (pressed_element)
            if (resize_handle == NO_HANDLE) {
               pressed_is_selected = False;
               for (i = 0; i < GetWindowInfoNumSelected(info); i++)
                  if (GetWindowInfoSelected(info, i) == pressed_element)
                     pressed_is_selected = True;
               if (!pressed_is_selected)
                  SetWindowInfoSelected(info, pressed_element);
            }
            else
               StartResize(pressed_element, &start_x, &start_y,
                           &last_x, &last_y);
         else
            DrawRubberband();
      }
      if (moved)
         if (pressed_element)
            if (resize_handle == NO_HANDLE)
               MoveAndDrawElements(info, x, y);
            else {
               DrawResizedGroup();
               Resize(pressed_element, resize_handle, x, y, &start_x, &start_y,
                      &last_x, &last_y);
               DrawResizedGroup();
            }
         else {
            DrawRubberband();
            last_x = x;
            last_y = y;
            DrawRubberband();
         }
   }
}

void DrawingExposeCallback(Widget w, XtPointer client, XtPointer call)
{
   InterfaceReDraw((WindowInfo *) client);
}
```

6.4 Graphics

6.4.1 graphics.h

```
/*
* Header
*/
/*
* Log
*/


#ifndef GRAPHICS_H
#define GRAPHICS_H

#include <Ixm.h>

Widget CreateMainForm(Widget);
```

```
#endif
```

6.4.2 graphics.c

```
/*
* Header
*/
/*
* Log
*/


#include <macros.h>  /* for min() and max() */
/*#include <malloc.h>*/
#include <stdio.h> /* TMP */
#include "drawing.h"
#include "commands.h"
#include "graphics.h"
#include "type.h"
#include "operations.h"
#include "visualize.h"
#include "pixmaps/mode/edit.xpm"
#include "pixmaps/mode/neuron.xpm"
#include "pixmaps/mode/connection.xpm"
#include "pixmaps/commands/group.xpm"
#include "pixmaps/commands/ungroup.xpm"
#include "pixmaps/commands/iconify.xpm"
#include "pixmaps/commands/deiconify.xpm"
#include "pixmaps/commands/delete.xpm"


#define DEFAULT_MODE EDIT_MODE
#define DEFAULT_ZOOM 1.0
#define DEFAULT_SHOW_CONNECTION TRUE
#define DEFAULT_SHOW_ID FALSE
#define OFFSET 10
#define BUTTON_OFFSET 10


static IxmMenubarStruct align_menu[] = {
   {"Left", 'L', LeftAlignCallback},
   {"Right", 'R', RightAlignCallback},
   {"Left-right-center", 'c', LeftRightCenterAlignCallback},
   {"", '-'},
   {"Top", 'T', TopAlignCallback},
   {"Bottom", 'B', BottomAlignCallback},
   {"Top-bottom-center", 'e', TopBottomCenterAlignCallback},
   {NULL}
};
static IxmMenubarStruct connection_menu[] = {
   {"Incoming", 'I', SelectIncomingConnectionsCallback},
   {"Outgoing", 'O', SelectOutgoingConnectionsCallback},
   {"Connecting", 'C', SelectConnectingConnectionsCallback},
   {NULL}
   };
static IxmMenubarStruct popup_menu[] = {
   {"Properties", 'P'},
   {"", '-'},
   {"Align", 'A', NULL, NULL, (IxmMenubarStruct *) align_menu},
   {"Adjust internals to group", 'i', AdjustInternalsCallback},
   {"Adjust group to internals", 'g', AdjustGroupCallback},
   {"", '-'},
   {"Select connections", 'S', NULL, NULL,
   (IxmMenubarStruct *) connection_menu},
   {"", '-'},
   {"Cut", 't', CutCallback},
   {"Copy", 'C', CopyCallback},
   {"Paste", 'P', PasteCallback},
   {"Delete", 'D', DeleteCallback},
   {"", '-'},
   {"Raise", 'R', RaiseCallback},
   {"Lower", 'L', LowerCallback},
   {NULL}
   };

static void SelectButton(WindowInfo *info, Mode m)
{
   static Pixel bottomshadow, topshadow;
   static Boolean first_time = True;
   Boolean edit_mode;


   /* Get the pixel values (only the first time) */
   if (first_time) {
      IxmSetArg(XmNtopShadowColor, &topshadow);
      IxmSetArg(XmNbottomShadowColor, &bottomshadow);
      IxmGetValues(GetWindowInfoButton(info, EDIT_MODE));
      first_time = False;
   }

   /* Deselect previously selected button */
   if (GetWindowInfoSelectedButton(info)) {
      IxmSetArg(XmNtopShadowColor, topshadow);
      IxmSetArg(XmNbottomShadowColor, bottomshadow);
      IxmSetValues(GetWindowInfoSelectedButton(info));
   }

   /* Select the new button */
   IxmSetArg(XmNtopShadowColor, bottomshadow);
   IxmSetArg(XmNbottomShadowColor, topshadow);
   IxmSetValues(GetWindowInfoButton(info, m));
   SetWindowInfoSelectedButton(info, GetWindowInfoButton(info, m));
}


void SetDefaultMode(WindowInfo *info)
{
   if (GetWindowInfoMode(info) != GetWindowInfoDefaultMode(info)) {
      SetWindowInfoMode(info, GetWindowInfoDefaultMode(info));
      SelectButton(info, GetWindowInfoMode(info));
   }
}


static void ButtonCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   int i;
   XmPushButtonCallbackStruct *cbs = (XmPushButtonCallbackStruct *) call;


   for (i = 0; (i < NUM_MODES) && (GetWindowInfoButton(info, i) != w); i++) ;

   /* Not possible */
   if (i == NUM_MODES)
      return;

   if (cbs->click_count == 1)
      SetWindowInfoMode(info, i);
   else
      SetWindowInfoDefaultMode(info, i);
   fprintf(stderr, "Mode = %d, Default = %d\n",
           (int) GetWindowInfoMode(info),
           (int) GetWindowInfoDefaultMode(info));
   SelectButton(info, i);
}


static void PopupCallback(Widget w, XtPointer client, XEvent *event,
                          Boolean *flag)
```

```
{
   WindowInfo *info = (WindowInfo *) client;


   if ((GetWindowInfoMode(info) == EDIT_MODE) &&
       (event->xbutton.button == Button3)) {
      SetWindowInfoMenuXPosition(info, event->xbutton.x);
      SetWindowInfoMenuYPosition(info, event->xbutton.y);
      *flag = True;
   }
   else
      *flag = False;
}


static void CreateButton(Widget parent, char *name, char *pixmap[],
                         Mode mode, WindowInfo *info)
{
   Widget w;


   /* Create the button */
   IxmSetArg(XmNhighlightThickness, 0);
   IxmSetArg(XmNshadowThickness, 2);
   IxmSetArg(XmNmultiClick, XmMULTICLICK_KEEP);
   w = IxmCreatePushButtonF(parent, name, ButtonCallback, (XtPointer) info);
   IxmSetLabelPixmap(w, pixmap);

   SetWindowInfoButton(info, mode, w);
}


static void ShowConnectionToggleCallback(Widget w, XtPointer client,
                                         XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   XmToggleButtonCallbackStruct *cbs = (XmToggleButtonCallbackStruct *) call;


   fprintf(stderr, "set: %d\n", /*XmToggleButtonGetState(w)*/ cbs->set);

   SetWindowInfoShowConnection(info, cbs->set);
   ReDraw(info);
}


static void ShowIDToggleCallback(Widget w, XtPointer client, XtPointer call)
{
   WindowInfo *info = (WindowInfo *) client;
   XmToggleButtonCallbackStruct *cbs = (XmToggleButtonCallbackStruct *) call;


   fprintf(stderr, "set: %d\n", /*XmToggleButtonGetState(w)*/ cbs->set);

   SetWindowInfoShowID(info, cbs->set);
   ReDraw(info);
}


Widget CreateMainForm(Widget parent)
{
   WindowInfo *info;
   Widget form, frame, draw, hori, verti;
   Widget main_rc, rc, w;
   IxmMenuRecord *rec;
   Dimension width, height;
   int i;


   /* Initlialize the WindowInfo structure */
   info = CreateWindowInfo(DEFAULT_MODE, DEFAULT_ZOOM, DEFAULT_SHOW_CONNECTION,
                                    DEFAULT_SHOW_ID);

    /* The main form */
    form = IxmCreateForm(parent);

    /* The rowcolumn on the right */
    IxmSetArg(XmNpacking, XmPACK_TIGHT);
    IxmSetArg(XmNnumColumns, 1);
    IxmSetArg(XmNspacing, OFFSET);
    main_rc = IxmCreateVerticalRowColumn(form);

    /* The frame for the rowcolumn with the buttons */
    frame = IxmCreateFrameWithTitle(main_rc, "Mode", XmSHADOW_ETCHED_IN);

    /* The rowcolumn with the buttons */
    IxmSetArg(XmNpacking, XmPACK_COLUMN);
    IxmSetArg(XmNnumColumns, 1);
    IxmSetArg(XmNspacing, BUTTON_OFFSET);
    rc = IxmCreateHorizontalRowColumn(frame);
    CreateButton(rc, "Edit", edit, EDIT_MODE, info);
    CreateButton(rc, "Neuron", neuron, NEURON_MODE, info);
    CreateButton(rc, "Connection", connection, CONNECTION_MODE, info);
    SelectButton(info, DEFAULT_MODE);

    /* The frame for the rowcolumn with the toggles */
    frame = IxmCreateFrameWithTitle(main_rc, "Viewing toggles",
                                    XmSHADOW_ETCHED_IN);

    /* The rowcolumn with the toggles */
    IxmSetArg(XmNpacking, XmPACK_COLUMN);
    IxmSetArg(XmNnumColumns, 1);
    rc = IxmCreateVerticalRowColumn(frame);
    IxmSetArg(XmNset, GetWindowInfoShowConnection(info));
    IxmCreateToggleButtonF(rc, "Show connections", ShowConnectionToggleCallback,
                           (XtPointer) info);
    IxmSetArg(XmNset, GetWindowInfoShowID(info));
    IxmCreateToggleButtonF(rc, "Show ID's", ShowIDToggleCallback,
                           (XtPointer) info);

    /* The frame for the rowcolumn with the commands */
    frame = IxmCreateFrameWithTitle(main_rc, "Commands", XmSHADOW_ETCHED_IN);

    /* The rowcollumn with the commands */
    IxmSetArg(XmNpacking, XmPACK_COLUMN);
    IxmSetArg(XmNnumColumns, 3);
    rc = IxmCreateVerticalRowColumn(frame);
    w = IxmCreatePushButtonF(rc, "Group", GroupCallback, (XtPointer) info);
    IxmSetLabelPixmap(w, group);
    SetWindowInfoCommandButton(info, GROUP_BUTTON, w);
    w = IxmCreatePushButtonF(rc, "Ungroup", UnGroupCallback, (XtPointer) info);
    IxmSetLabelPixmap(w, ungroup);
    SetWindowInfoCommandButton(info, UNGROUP_BUTTON, w);
    w = IxmCreatePushButtonF(rc, "Iconify", IconifyCallback, (XtPointer) info);
    IxmSetLabelPixmap(w, iconify);
    SetWindowInfoCommandButton(info, ICON_BUTTON, w);
    w = IxmCreatePushButtonF(rc, "DeIconify", DeIconifyCallback,
                             (XtPointer) info);
    IxmSetLabelPixmap(w, deiconify);
    SetWindowInfoCommandButton(info, DEICON_BUTTON, w);
    w = IxmCreatePushButtonF(rc, "Delete", DeleteCallback,
                             (XtPointer) info);
    IxmSetLabelPixmap(w, delete);
    SetWindowInfoCommandButton(info, DELETE_BUTTON, w);


    /* The frame for the rowcolumn with the dialogs */
    frame = IxmCreateFrameWithTitle(main_rc, "Dialogs", XmSHADOW_ETCHED_IN);

    /* The scrollbars */
    verti = IxmCreateVerticalScrollBar(form);
    hori = IxmCreateHorizontalScrollBar(form);
    IxmSetArg(XmNwidth, &width);
    IxmGetValues(verti);
    IxmSetArg(XmNheight, &height);
    IxmGetValues(hori);

    /* The frame & drawingarea */
    frame = IxmCreateFrame(form, XmSHADOW_IN);
    IxmSetArg(XmNbackground,
              BlackPixel(Ixm_display, DefaultScreen(Ixm_display)));
    draw = IxmCreateDrawingArea(frame, 800, 400);
    IxmAddCallback(draw, XmNexposeCallback, DrawingExposeCallback,
                   (XtPointer) info, 0);
    IxmAddEventHandlerF(draw, ButtonPressMask, DrawingButtonCallback,
                        (XtPointer) info);
    IxmAddEventHandlerF(draw, ButtonReleaseMask, DrawingButtonCallback,
                        (XtPointer) info);
    IxmAddEventHandlerF(draw, ButtonMotionMask, DrawingMotionCallback,
                        (XtPointer) info);

    /* Set the constraints and offsets */
    IxmSetConstraints(main_rc, XmATTACH_FORM, XmATTACH_FORM, XmATTACH_NONE,
                      XmATTACH_FORM, NULL, NULL, NULL, NULL);
    IxmSetConstraints(verti, XmATTACH_FORM, XmATTACH_OPPOSITE_WIDGET,
                      XmATTACH_NONE, XmATTACH_WIDGET, NULL, frame, NULL,
                      main_rc);
    IxmSetConstraints(hori, XmATTACH_NONE, XmATTACH_FORM, XmATTACH_FORM,
                      XmATTACH_OPPOSITE_WIDGET, NULL, NULL, NULL, frame);
    IxmSetConstraints(frame, XmATTACH_FORM, XmATTACH_FORM, XmATTACH_FORM,
                      XmATTACH_WIDGET, NULL, NULL, NULL, main_rc);
    IxmSetOffsets(main_rc, OFFSET, OFFSET, IxmNoOffset, OFFSET);
    IxmSetOffsets(verti, OFFSET, 0, IxmNoOffset, OFFSET);
    IxmSetOffsets(hori, IxmNoOffset, OFFSET, OFFSET, 0);
    IxmSetOffsets(frame, OFFSET, 2 * OFFSET + height,
                  OFFSET, 2 * OFFSET + width);

    /* Create popupmenu */
    for (i = 0; connection_menu[i].namestr; i++)
       connection_menu[i].data = (int *) &info;
    for (i = 0; align_menu[i].namestr; i++)
       align_menu[i].data = (int *) &info;
    for (i = 0; popup_menu[i].namestr; i++)
       popup_menu[i].data = (int *) &info;
    rec = IxmCreatePopupMenu(draw, (IxmMenubarStruct *) popup_menu,
                             PopupCallback, (XtPointer) info);

    /* Set the attached widgets */
    SetWindowInfoDrawingArea(info, draw);
    SetWindowInfoHorizontalScrollBar(info, hori);
    SetWindowInfoVerticalScrollBar(info, verti);

    /* Initialize the visualization stuff */
    ClassInitialize();

    /* Finally .. Give the scrollbars the correct parameters */
    /*
    SetScrollBars(info);
    */

    /*
    XSynchronize(Ixm_display, 1);
    */

    return(form);
}
```

6.5 Interface

6.5.1 interface.h

```
#include <Ixm.h>
```

```
#include "type.h"


void InterfaceReDraw(WindowInfo *info);
void InterfaceIxmDrawLine(Widget, GC, int, int, int, int);
void InterfaceIxmDrawRectangle(Widget, GC, int, int, int, int);
void InterfaceIxmFillRectangle(Widget, GC, int, int, int, int);
void InterfaceIxmDrawArc(Widget, GC, int, int, int, int, int, int);
void InterfaceIxmFillArc(Widget, GC, int, int, int, int, int, int);
void InterfaceIxmDrawString(Widget, GC, int, int, char *str);
```

6.5.2 interface.c

```
#ifdef HANS
#include "Graph.h"
#endif
#include "interface.h"
#include "visualize.h"


void InterfaceReDraw(WindowInfo *info)
{
#ifdef HANS
   GetWindowInfoDrawingArea (info)->core.parent->core.widget_class->core_clas
      s.expose (GetWindowInfoDrawingArea (info)->core.parent, NULL, NULL);
#else
   ReDraw(info);
#endif
}


void InterfaceIxmDrawLine(Widget w, GC gc, int x1, int y1, int x2, int y2)
{
#ifdef HANS
   IxmDrawLine(w, gc, x1, y1, x2, y2);
#else
   XDrawLine(Ixm_display, XtWindow(w), gc, x1, y1, x2, y2);
#endif
}


void InterfaceIxmDrawRectangle(Widget w, GC gc, int x, int y, int width,
                               int height)
{
#ifdef HANS
   IxmDrawRectangle(w, gc, x, y, width, height);
#else
   XDrawRectangle(Ixm_display, XtWindow(w), gc, x, y, width, height);
#endif
}


void InterfaceIxmFillRectangle(Widget w, GC gc, int x, int y, int width,
                               int height)
{
#ifdef HANS
   IxmFillRectangle(w, gc, x, y, width, height);
#else
   XFillRectangle(Ixm_display, XtWindow(w), gc, x, y, width, height);
#endif
}


void InterfaceIxmDrawArc(Widget w, GC gc, int x, int y, int width, int height,
                         int angle1, int angle2)
{
#ifdef HANS
   IxmDrawArc(w, gc, x, y, width, height, angle1, angle2);
#else
   XDrawArc(Ixm_display, XtWindow(w), gc, x, y, width, height, angle1, angle2);
#endif
}


void InterfaceIxmFillArc(Widget w, GC gc, int x, int y, int width, int height,
                         int angle1, int angle2)
{
#ifdef HANS
   IxmFillArc(w, gc, x, y, width, height, angle1, angle2);
#else
   XFillArc(Ixm_display, XtWindow(w), gc, x, y, width, height, angle1, angle2);
#endif
}


void InterfaceIxmDrawString(Widget w, GC gc, int x, int y, char *str)
{
#ifdef HANS
   IxmDrawString(w, gc, x, y, str);
#else
   XDrawString(Ixm_display, XtWindow(w), gc, x, y, str, strlen(str));
#endif
}
```

6.6 Main

6.6.1 main.c

```
/*
* Header
*/
/*
* Log
*/
#include "graphics.h"


static String fallback[] = {
   /*
   "*fontList: -*-helvetica-bold-r-*-*-*-120-*-*-*-*-iso8859-*,\
   */
   "*fontList: -*-helvetica-bold-r-*-*-12-120-*-*-*-*-iso8859-*,\
   -*-helvetica-bold-r-*-*-*-240-*=version_font,\
   -*-helvetica-medium-r-*-*-*-120-*=copyright_font ",
   "*XmTextField.fontList: -*-courier-medium-r-*-*-*-120-*-*-*-*-iso8859-*",
   "*XmText.fontList: -*-courier-medium-r-*-*-*-120-*-*-*-*-iso8859-*",
   "*XmList*fontList: -*-courier-medium-r-*-*-*-120-*-*-*-*-iso8859-*",
   NULL};


void QuitCallback(Widget w, XtPointer client, XtPointer call)
{
   exit(0);
}


void main(int argc, char **argv)
{
   XtAppContext app;
   Widget toplevel, shell, form, form2, but; /* pane */

   /* Create widgets and initialize Ixm */
   toplevel = XtVaAppInitialize(&app, "Black Hole", NULL, 0, &argc, argv,
                                fallback, XmNmappedWhenManaged, FALSE, NULL);
   XtRealizeWidget(toplevel);
   IxmStartup();
   IxmInitialize(toplevel);
   shell = IxmCreateWindow(toplevel, "Testje", "Icoontje");
   /* pane = IxmCreatePanedWindow(shell); */
   form = IxmCreateForm(shell);
   /* Create the three subwidgets */
   /*   IxmCreateLabel(pane, "Top");
   CreateMainForm(pane);
   IxmCreateLabel(pane, "Bottom");
   IxmCreatePushButton(pane, "Quit", QuitCallback, 0);
   */
   form2 = CreateMainForm(form);
   but = IxmCreatePushButton(form, "Quit", QuitCallback, 0);
   IxmSetConstraints(form2, XmATTACH_FORM, XmATTACH_WIDGET, XmATTACH_FORM,
                     XmATTACH_FORM, NULL, but, NULL, NULL);
   IxmSetConstraints(but, XmATTACH_NONE, XmATTACH_FORM, XmATTACH_FORM,
                     XmATTACH_FORM, NULL, NULL, NULL, NULL);


   /* Main loop */
   XtPopup(shell, XtGrabNone);
   XtAppMainLoop(app);
}
```

6.7 Operations

6.7.1 operations.h

```
/*
* Header
*/
/*
* Log
*/

#ifndef OPERATIONS_H
#define OPERATIONS_H


#include "constants.h"
#include "type.h"


typedef enum {
   TOP_LEFT, BOTTOM_LEFT, TOP_RIGHT, BOTTOM_RIGHT, NO_HANDLE
} ResizeHandle;
typedef enum {
   LEFT, RIGHT, LEFT_RIGHT_CENTER,
   TOP, BOTTOM, TOP_BOTTOM_CENTER
} Alignment;
typedef enum {
   INCOMING, OUTGOING, CONNECTING
} SelectConnectionType;


#define InRange(x, x1, x2) \
   ((x1 <= x) && (x <= x2))
#define CoordinatesInRange(x, y, x1, y1, x2, y2) \
   ((InRange(x, x1, x2) || InRange(x, x2, x1)) &&\
   (InRange(y, y1, y2) || InRange(y, y2, y1)))
#define LeftPosition(e) (GetElementXPosition(e) - ExternalWidth(e) / 2)
#define RightPosition(e) (GetElementXPosition(e) + (ExternalWidth(e) + 1) / 2)
#define TopPosition(e) (GetElementYPosition(e) - ExternalHeight(e) / 2)
#define BottomPosition(e) (GetElementYPosition(e) + (ExternalHeight(e) + 1) / 2)
#define ExternalWidth(e) \
   ((GetElementType(e) == NEURON_TYPE) ? NEURON_WIDTH : \
   (((GetElementType(e) == GROUP_TYPE) && !GetGroupOpened(e)) ? \
   GROUP_WIDTH : GetElementWidth(e)))
#define ExternalHeight(e) \
   ((GetElementType(e) == NEURON_TYPE) ? NEURON_WIDTH : \
   (((GetElementType(e) == GROUP_TYPE) && !GetGroupOpened(e)) ? \
   GROUP_HEIGHT : GetElementHeight(e)))
```

```
#define InElement(x, y, e) \
   (CoordinatesInRange(x, y, LeftPosition(e), TopPosition(e), \
   RightPosition(e), BottomPosition(e)))
#define IsVisible(e) ((e) == ClosedParent((e)))

Boolean ShowConnection(Element*);
void ProcessPositionAndDimension(Element*);
void GetPositionAndDimension(Position*, Position*, Dimension*, Dimension*);
Element *GetElementWithinRange(WindowInfo*, Position, Position, ElementTypeMask,
                               ElementTypeMask);
Element *GetElementInEditLevel(WindowInfo*, Position, Position, ElementTypeMask,
                               ElementTypeMask);
Element *GetConnectionAtPosition(WindowInfo*, Position, Position);
ResizeHandle GetResizeHandle(WindowInfo*, int, int, Element**);
void StartResize(Element*, int*, int*, int*, int*);
void Resize(Element*, ResizeHandle, int, int, int*, int*, int*, int*);
void RecursiveSetConnectionPositions(Element*);
void SetConnectionPositions(Element*);
Element *AddNeuron(WindowInfo*, int, int);
void Connect(WindowInfo*, Element*, Element*);
Boolean PropagateSizes(Element*);
void RecursiveMoveElement(Element*, int, int);
Element *ClosedParent(Element*);
void NewSelected(WindowInfo*, Element*);
void AlignElements(WindowInfo*, Alignment);
void SelectConnections(WindowInfo*, SelectConnectionType);
Element *GetConnectionEnd(WindowInfo*, int, int);
void EnterGroup(WindowInfo*, Element*);
void LeaveGroup(WindowInfo*, Element*);
void Cut(WindowInfo*);
void Copy(WindowInfo*);
void Paste(WindowInfo*);

#endif
```

6.7.2 operations.c

```
/*
* Header
*/
/*
* Log
*/


#include <macros.h>
/*#include <malloc.h>*/
#include <math.h>
#include <stdio.h> /* TMP */
#include "constants.h"
#include "operations.h"
#include "visualize.h"  /* for DrawSelected() */


/* These two structures are used for making a list of elements and their
   copies. This is necessary for defining the new connections */
typedef struct {
   Element *source, *copy;
} CopyListData;
typedef struct {
   CopyListData *list;
   int num;
} CopyList;


/* Static variables used by ProcessPositionsAndDimension(),
   GetPositionAndDimension() and GetPositions()
   */
static int min_x, min_y, max_x, max_y, num = 0;
```

```
/* Initialize-routine for the list of copies
   */
static void InitList(CopyList *list)
{
   list->num = 0;
   list->list = NULL;
}


/* Routine for freeing the list of copies
   */
static void FreeList(CopyList *list)
{
   free(list->list);
}


/* Routine for adding a new source and copy to the list
   */
static void AddCopy(CopyList *list, Element *source, Element *copy)
{
   list->list =
      (CopyListData *) realloc(list->list,
                               (list->num + 1) * sizeof(CopyListData));
   list->list[list->num].source = source;
   list->list[list->num].copy = copy;
   list->num++;
}


/* Routine for extracting a copy of an element from the list. If
   no copy was found, the element itself will be returned
   */
static Element *GetCopy(CopyList *list, Element *source)
{
   int i;


   /* Find the source */
   for (i = 0; (i < list->num) && (list->list[i].source != source); i++) ;

   if (i == list->num)
      return(source);  /* Source was not found in the list */
   else
      return(list->list[i].copy);  /* Source was found in the list,
                                      so return the copy */
}


/* Utility for determining the top-left (x,y) and bottom-right (x,y)
   of some elements
   */
void ProcessPositionAndDimension(Element *e)
{
   if (!num) {
      min_x = LeftPosition(e);
      max_x = RightPosition(e);
      min_y = TopPosition(e);
      max_y = BottomPosition(e);
   }
   else {
      min_x = min(min_x, LeftPosition(e));
      max_x = max(max_x, RightPosition(e));
      min_y = min(min_y, TopPosition(e));
      max_y = max(max_y, BottomPosition(e));
   }
   num++;
}
```

```
/* Called after using ProcessPositionAndDimension() to retrieve
   the middle (x,y), width and height of some elements
   */
void GetPositionAndDimension(Position *x, Position *y,
                             Dimension *w, Dimension *h)
{
   *x = (min_x + max_x) / 2;
   *y = (min_y + max_y) / 2;
   *w = max_x - min_x;
   *h = max_y - min_y;
   num = 0;
}


/* Called after using ProcessPositionAndDimension() to retrieve
   the top-left (x,y) and bottom-right(x,y) of some elements
   */
void GetPositions(Position *x1, Position *y1, Position *x2, Position *y2)
{
   *x1 = min_x;
   *y1 = min_y;
   *x2 = max_x;
   *y2 = max_y;
   num = 0;
}


/* Function to determine whether a connection has to be visualized.
   */
Boolean ShowConnection(Element *connection)
{
   ConnectionState state = GetConnectionState(connection);
   Element *element;


   /* If the connectionstate is default return the default
      connectionstate */
   if (state == DEFAULT)
      return(GetWindowInfoShowConnection(GetElementInfo(element)));

   /* If state is parent, search the connectionstate from the parents
      and put the value in "state" */
   if (state == PARENT) {
      /* Get the parent-group */
      element = GetNeuronParent(GetConnectionFrom(connection));

      /* Search until a parent is found that has its own connectionstate
         or until the main group is found */
      while ((GetGroupConnectionState(element) == PARENT) &&
             !GetGroupMainGroup(element))
         element = GetGroupParent(element);

      if ((GetGroupConnectionState(element) == PARENT) ||
          (GetGroupConnectionState(element) == DEFAULT))
         /* Arrived at the maingroup with connectionstate parent or
            arrived at another group with connectionstate default */
         return(GetWindowInfoShowConnection(GetElementInfo(element)));
   }

   /* "state" is now VISIBLE or INVISIBLE */
   return((state == VISIBLE) ? TRUE : FALSE);
}


/* Function to retrieve an element at a specified position. Parameters
   are used to define searchmasks of elementtypes that have to be returned
   or not
   */
Element *GetElementWithinRange(WindowInfo *info, Position x, Position y,
                               ElementTypeMask search,
                               ElementTypeMask no_search)
```

```
{
   Element *e;
   int i;


   /* Search in reverse order so the last added element will
      be found first */
   for (i = GetWindowInfoNumElements(info) - 1; (i >= 0); i--) {
      e = GetWindowInfoElement(info, i);

      /* If the element is at the specified position, the search masks are
         ok and the element is visible, return the element */
      if (InElement(x, y, e) && (search & (1 << GetElementType(e))) &&
          !(no_search & (1 << GetElementType(e))) && IsVisible(e))
         return(e);
   }
   return(NULL);
}


/* Function to retrieve an element at a specified position within the
   current editlevel. Parameters are used to define searchmasks of
   elementtypes that have to be returned or not
   */
Element *GetElementInEditLevel(WindowInfo *info, Position x, Position y,
                               ElementTypeMask search,
                               ElementTypeMask no_search)
{
   Element *edit = GetWindowInfoEditLevel(info);
   Element *e;
   int i;


   /* Try to find a child within the edit level, search reverse so
      the child last added will be found first */
   for (i = GetGroupNumInternals(edit) - 1; (i >= 0); i--) {
      e = GetGroupInternal(edit, i);
      /* If the element is at the specified position, the search masks are
         ok and the element is visible, return the element */
      if (InElement(x, y, e) && (search & (1 << GetElementType(e))) &&
          !(no_search & (1 << GetElementType(e))) && IsVisible(e))
         return(e);
   }
   return(NULL);
}


/* Function to find a connection at position (x,y)
   */
Element *GetConnectionAtPosition(WindowInfo *info, Position x, Position y)
{
   Element *e;
   int i;
   int x1, y1, dx, dy;
   float lambda;


   /* Search the elements in reverse order so the last added connection
      will be found first */
   for (i = GetWindowInfoNumElements(info) - 1; (i >= 0); i--) {
      e = GetWindowInfoElement(info, i);
      if (GetElementType(e) == CONNECTION_TYPE) {

         /* If this is a connection from and to the same neuron, just
            check the bounding box of the connection (top-, bottom-, left-
            and right-position) */
         if (GetConnectionFrom(e) == GetConnectionTo(e))
            if (CoordinatesInRange(x, y, LeftPosition(e), TopPosition(e),
                                   RightPosition(e), BottomPosition(e)))
               return(e);
            else
```

```
               continue;

         /* Get the starting position and the horizontal and vertical
            distance of the connection */
         x1 = GetConnectionFromXPosition(e);
         y1 = GetConnectionFromYPosition(e);
         dx = GetConnectionToXPosition(e) - x1;
         dy = GetConnectionToYPosition(e) - y1;

         /* skip if the connection has no length */
         if (!dx && !dy)
            continue;

         /* Determine the position of the crossing of the perpendicular of the
            connection through (x,y) and the connection itself */
         lambda = 1.0 * (dx * x + dy * y -
                         dx * x1 - dy * y1) / (dx * dx + dy * dy);

         /* If the perpendicular doesn't cross the connection, skip */
         if ((lambda < 0.0) || (lambda > 1.0))
            continue;

         /* If the distance is small enough, return this connection */
         if (sqrt(pow(x1 + lambda * dx - x, 2) +
                  pow(y1 + lambda * dy - y, 2)) < CONNECTION_DISTANCE)
            return(e);
      }
   }

   /* Nothing found, so return nothing */
   return(NULL);
}


/* Function to determine whether the mouse is pressed at a corner of an
   opened group. If not, NO_HANDLE is returned. If a corner was pressed,
   TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT or BOTTOM_RIGHT is returned. Element **e
   will be filled with the group.
   */
ResizeHandle GetResizeHandle(WindowInfo *info, int x, int y, Element **e)
{
   int i;


   /* Search the elements in reverse order so the last added connection
      will be found first */
   for (i = GetWindowInfoNumSelected(info) - 1; (i >= 0); i--) {
      *e = GetWindowInfoSelected(info, i);

      /* Check whether a resize-handle is pressed. The top-handle is located
         at (LeftPosition(*e) - SELECTED_PADDING, TopPosition(*e) -
         SELECTED_PADDING) and has a diameter of SELECTED_WIDTH */
      if ((GetElementType(*e) == GROUP_TYPE) && GetGroupOpened(*e))
         if (CoordinatesInRange(x, y, LeftPosition(*e) - SELECTED_PADDING -
                               SELECTED_WIDTH / 2, TopPosition(*e) -
                               SELECTED_PADDING - SELECTED_WIDTH / 2,
                               LeftPosition(*e) - SELECTED_PADDING +
                               SELECTED_WIDTH / 2, TopPosition(*e) -
                               SELECTED_PADDING + SELECTED_WIDTH / 2))
            return(TOP_LEFT);
         else if (CoordinatesInRange(x, y, LeftPosition(*e) - SELECTED_PADDING -
                                    SELECTED_WIDTH / 2, BottomPosition(*e) +
                                    SELECTED_PADDING - SELECTED_WIDTH / 2,
                                    LeftPosition(*e) - SELECTED_PADDING +
                                    SELECTED_WIDTH / 2, BottomPosition(*e) +
                                    SELECTED_PADDING + SELECTED_WIDTH / 2))
            return(BOTTOM_LEFT);
         else if (CoordinatesInRange(x, y, RightPosition(*e) + SELECTED_PADDING -
                                    SELECTED_WIDTH / 2, TopPosition(*e) -
                                    SELECTED_PADDING - SELECTED_WIDTH / 2,
                                    RightPosition(*e) + SELECTED_PADDING +
                                        SELECTED_WIDTH / 2, TopPosition(*e) -
                                        SELECTED_PADDING + SELECTED_WIDTH / 2))
            return(TOP_RIGHT);
         else if (CoordinatesInRange(x, y, RightPosition(*e) + SELECTED_PADDING -
                                        SELECTED_WIDTH / 2, BottomPosition(*e) +
                                        SELECTED_PADDING - SELECTED_WIDTH / 2,
                                        RightPosition(*e) + SELECTED_PADDING +
                                        SELECTED_WIDTH / 2, BottomPosition(*e) +
                                        SELECTED_PADDING + SELECTED_WIDTH / 2))
            return(BOTTOM_RIGHT);
   }

   /* Nothing found, so return no handle */
   return(NO_HANDLE);
}


/* This routine is called when starting to resize. The four integers
   determining the corners of the element are initialized
   */
void StartResize(Element *e, int *x1, int *y1, int *x2, int *y2)
{
   *x1 = LeftPosition(e);
   *y1 = TopPosition(e);
   *x2 = LeftPosition(e) + GetElementWidth(e);
   *y2 = TopPosition(e) + GetElementHeight(e);
}


/* Routine to resize an element. The parameters define the group, the
   resize-handle, the current mouse-position and the corners after the last
   resize of the group. The group can't become smaller than its contents.
   */
void Resize(Element *group, ResizeHandle h, int x, int y,
            int *x1, int *y1, int *x2, int *y2)
{
   Element *e;
   int i;


   /* Define the new x1, y1, x2 and y2. Depending on the value of
      ResizeHandle h, two of these values are changed */
   switch(h) {
      case TOP_LEFT:
         *x1 = x;
         *y1 = y;
         break;
      case TOP_RIGHT:
         *x2 = x;
         *y1 = y;
         break;
      case BOTTOM_LEFT:
         *x1 = x;
         *y2 = y;
         break;
      case BOTTOM_RIGHT:
         *x2 = x;
         *y2 = y;
         break;
   }

   /* The group can't become smaller than its contents */
   for (i = 0; i < GetGroupNumInternals(group); i++) {
      e = GetGroupInternal(group, i);
      *x1 = min(*x1, LeftPosition(e) - GROUP_PADDING);
      *x2 = max(*x2, RightPosition(e) + GROUP_PADDING);
      *y1 = min(*y1, TopPosition(e) - GROUP_PADDING);
      *y2 = max(*y2, BottomPosition(e) + GROUP_PADDING);
   }
}
```

```
/* Routine to retrieve the correct begin- and end-positions of a connection
   between two different elements. Connections can be attached to elements
   in different ways, depending on the orientation of the connection. If
   the horizontal distance is bigger than the vertical distance, the oriention
   is horizontal, otherwise vertical.
   */
static void GetConnectionPositions(Element *e, int *x1, int *y1,
                                   int *x2, int *y2)
{
   Element *from = ClosedParent(GetConnectionFrom(e));
   Element *to = ClosedParent(GetConnectionTo(e));


   *x1 = GetElementXPosition(from);
   *y1 = GetElementYPosition(from);
   *x2 = GetElementXPosition(to);
   *y2 = GetElementYPosition(to);
   if (abs(*x1 - *x2) > abs(*y1 - *y2))
      /* horizontal orientated line */
      if (*x1 < *x2) {
         *x1 = RightPosition(from);
         *x2 = LeftPosition(to);
      }
      else {
         *x1 = LeftPosition(from);
         *x2 = RightPosition(to);
      }
   else
      /* vertical orientated line */
      if (*y1 < *y2) {
         *y1 = BottomPosition(from);
         *y2 = TopPosition(to);
      }
      else {
         *y1 = TopPosition(from);
         *y2 = BottomPosition(to);
      }
}


/* Routine to the correct begin- and end-position of a connection
   */
void SetConnectionPositions(Element *e)
{
   int x1, y1, x2, y2;


   if (GetConnectionFrom(e) != GetConnectionTo(e)) {
      /* The connected elements are different, so the connectionpositions are
         retrieved using GetConnectionPositions() */
      GetConnectionPositions(e, &x1, &y1, &x2, &y2);
      SetConnectionFromXPosition(e, x1);
      SetConnectionFromYPosition(e, y1);
      SetConnectionToXPosition(e, x2);
      SetConnectionToYPosition(e, y2);
   }
   else {
      Element *from = GetConnectionFrom(e);

      /* The connected elements are the same, so the correct (x,y)-position
         of the element has to be defined */
      SetElementXPosition(e, GetElementXPosition(from));
      SetElementYPosition(e, GetElementYPosition(from) - NEURON_WIDTH);
   }
}


/* Routine to set all the connectionpositions of the childs of an element
   */
void RecursiveSetConnectionPositions(Element *e)
```

```
{
   int i;


   if (GetElementType(e) == GROUP_TYPE)
      for (i = 0; i < GetGroupNumInternals(e); i++)
         RecursiveSetConnectionPositions(GetGroupInternal(e, i));
   if (GetElementType(e) == NEURON_TYPE)
      for (i = 0; i < GetNeuronNumConnections(e); i++)
         SetConnectionPositions(GetNeuronConnection(e, i));
}


/* Routine to add a new neuron. The neuron must be added within the range of
   the current editlevel, of nothing is done. After creating the new neuron,
   the neuron is added in the list of childs of the current editlevel
   */
Element *AddNeuron(WindowInfo *info, int x, int y)
{
   Element *new, *edit = GetWindowInfoEditLevel(info);


   if ((edit == GetWindowInfoMainGroup(info)) ||
        CoordinatesInRange(x, y, LeftPosition(edit) +
                          GROUP_PADDING + NEURON_WIDTH / 2,
                          TopPosition(edit) + GROUP_PADDING +
                          NEURON_WIDTH / 2, RightPosition(edit) -
                          GROUP_PADDING - NEURON_WIDTH / 2,
                          BottomPosition(edit) - GROUP_PADDING -
                          NEURON_WIDTH / 2)) {
      new = CreateNeuron(edit, x, y);
      AddGroupInternal(edit, new);
      return(new);
   }
   else
      return(NULL);
}


/* Utility-routine to connect two elements. If one of the elements is an
   opened group, all the internals will be connected.
   */
static void _Connect(WindowInfo *info, Element *from, Element *to)
{
   Element *e;
   int i;


   /* If a group is connected to itself, skip.
      Maybe this line can be deleted? */
   if ((from == to) && (GetElementType(from) == GROUP_TYPE))
      return;

   if ((GetElementType(from) == GROUP_TYPE) && GetGroupOpened(from))
      /* Recursively connect the internals */
      for (i = 0; i < GetGroupNumInternals(from); i++)
         _Connect(info, GetGroupInternal(from, i), to);
   else if ((GetElementType(to) == GROUP_TYPE) && GetGroupOpened(to))
      /* Recursively connect the internals */
      for (i = 0; i < GetGroupNumInternals(to); i++)
         _Connect(info, from, GetGroupInternal(to, i));
   else if ((GetElementType(from) == NEURON_TYPE) &&
            (GetElementType(to) == NEURON_TYPE)) {
      /* Connect the neurons, set the connectionpositions and make the new
         connections selected */
      if (!(e = CreateConnection(from, to, PARENT)))
         return;
      SetConnectionPositions(e);
      AddWindowInfoSelected(info, e);
   }
}
```

```
/* Routine to connect two elements. Elements can be opened groups or neurons.
   Connections from an opened group to itself are ignored (see _Connect() above)
   */
void Connect(WindowInfo *info, Element *from, Element *to)
{
   int i;


   /* Connect the elements */
   _Connect(info, from, to);

   /* Draw the new connections and the selection corners */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      DrawElement(GetWindowInfoSelected(info, i), False);
   DrawCorners(info);
}


/* Function to check whether the internal size of a group has changed. If
   the size has changed, the same routine will be used to check the size
   of its parent. The size can only become bigger.
   */
Boolean PropagateSizes(Element *e)
{
   int i;
   int min_x, min_y, max_x, max_y;
   Boolean b = False;


   /* Skip if we are not dealing with a group, the element is the main group
      or the group contains no internals */
   if ((GetElementType(e) != GROUP_TYPE) ||
       GetGroupMainGroup(e) || !GetGroupNumInternals(e))
      return(False);

   /* Initialize the minimal and maximal (x,y) of the internals */
   min_x = LeftPosition(e) + GROUP_PADDING;
   max_x = RightPosition(e) - GROUP_PADDING;
   min_y = TopPosition(e) + GROUP_PADDING;
   max_y = BottomPosition(e) - GROUP_PADDING;

   /* Retrieve the total minimal and maximal (x,y) of the internals */
   for (i = 0; i < GetGroupNumInternals(e); i++) {
      min_x = min(min_x, LeftPosition(GetGroupInternal(e, i)));
      max_x = max(max_x, RightPosition(GetGroupInternal(e, i)));
      min_y = min(min_y, TopPosition(GetGroupInternal(e, i)));
      max_y = max(max_y, BottomPosition(GetGroupInternal(e, i)));
   }

   /* If the internals occupy more space than the width of the group, resize
      the group. Set the flag True */
   if ((max_x - min_x + GROUP_PADDING * 2) > GetElementWidth(e)) {
      SetElementXPosition(e, (min_x + max_x) / 2);
      SetElementWidth(e, max_x - min_x + 2 * GROUP_PADDING);
      b = True;
   }

   /* Dito for height */
   if ((max_y - min_y + GROUP_PADDING * 2) > GetElementHeight(e)) {
      SetElementYPosition(e, (min_y + max_y) / 2);
      SetElementHeight(e, max_y - min_y + 2 * GROUP_PADDING);
      b = True;
   }

   /* If the size has changed, check the parent */
   if (b)
      PropagateSizes(GetElementParent(e));
   return(b);
}


/* Routine to move an element. If the element is a neuron, the new connection-
   positions are determined. If the element is a group, the internals are moved
   using the same routine.
   */
void RecursiveMoveElement(Element *e, int dx, int dy)
{
   int i;


   SetElementXPosition(e, GetElementXPosition(e) + dx);
   SetElementYPosition(e, GetElementYPosition(e) + dy);
   if (GetElementType(e) == NEURON_TYPE)
      for (i = 0; i < GetNeuronNumConnections(e); i++)
         SetConnectionPositions(GetNeuronConnection(e, i));
   if (GetElementType(e) == GROUP_TYPE)
      for (i = 0; i < GetGroupNumInternals(e); i++)
         RecursiveMoveElement(GetGroupInternal(e, i), dx, dy);
}


/* Function to find a closed parent of a neuron, which is the "highest"
   in the hierarchy. This is the element visible for the user, containing
   this neuron.
   */
Element *ClosedParent(Element *e)
   /* Only for neurons */
{
   Element *parent = GetNeuronParent(e);
   Element *return_val = e;


   while (!GetGroupMainGroup(parent)) {
      if (!GetGroupOpened(parent))
         return_val = parent;
      parent = GetGroupParent(parent);
   }

   return(return_val);
}


/* A new item is selected. First remove the corners of the old selected item(s)
   by drawing them again (using the XOR), define the new selected item and
   draw the corners again.
   */
void NewSelected(WindowInfo *info, Element *e)
{
   DrawCorners(info);
   SetWindowInfoSelected(info, e);
   DrawSelected(info);
}


/* Function which returns True if an element is selected and False otherwise
   */
static Boolean IsSelected(WindowInfo *info, Element *e)
{
   int i;


   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      if (GetWindowInfoSelected(info, i) == e)
         return(True);
   return(False);
}
```

```
/* Function to align selected elements.
   */
void AlignElements(WindowInfo *info, Alignment align)
{
   Element *e;
   Position x1, y1, x2, y2;
   int i, j;


   /* Only continue if there are elements selected */
   if (!GetWindowInfoNumSelected(info))
      return;

   /* Determine the left, top, right and bottom position of the elements */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      ProcessPositionAndDimension(GetWindowInfoSelected(info, i));
   GetPositions(&x1, &y1, &x2, &y2);

   /* Align all the selected elements */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
      e = GetWindowInfoSelected(info, i);
      switch(align) {
         case TOP:
            RecursiveMoveElement(e, 0, y1 - TopPosition(e));
            break;
         case BOTTOM:
            RecursiveMoveElement(e, 0, y2 - BottomPosition(e));
            break;
         case TOP_BOTTOM_CENTER:
            RecursiveMoveElement(e, 0, (y1 + y2) / 2 - GetElementYPosition(e));
            break;
         case LEFT:
            RecursiveMoveElement(e, x1 - LeftPosition(e), 0);
            break;
         case RIGHT:
            RecursiveMoveElement(e, x2 - RightPosition(e), 0);
            break;
         case LEFT_RIGHT_CENTER:
            RecursiveMoveElement(e, (x1 + x2) / 2 - GetElementXPosition(e), 0);
            break;
      }
   }
}


/* Utility routine to make a list of connections. An element is added to the
   list if it doesn't occur in the list yet. The integer num indicates the
   length of the list.
   */
static void AddNewConnection(Element ***connection_list, int *num,
                             Element *e)
{
   int i;


   for (i = 0; i < *num; i++)
      if ((*connection_list)[i] == e)
         return;

   *connection_list = (Element **) realloc(*connection_list,
                                           (*num + 1) * sizeof(Element *));
   (*connection_list)[*num] = e;
   (*num)++;
}


/* Recursive select connections of the selected elements. The
   SelectConnectionType "type" can be INCOMING, OUTGOING or CONNECTING.
   CONNECTING will select all the connections that connect two or more
   selected elements. "selected" is the element that was selected and
   "e" is the element that has to be checked (e == selected or e is a
   child of selected). Connections that has to be selected are added in
   the "connection_list".
   */
static void RecursiveSelectConnections(WindowInfo *info, Element *selected,
                                       Element *e, SelectConnectionType type,
                                       Element ***connection_list, int *num)
{
   Element *connection, *other, *parent;
   int i;


   if (GetElementType(e) == GROUP_TYPE)
      /* Recursively select the connections of the internals */
      for (i = 0; i < GetGroupNumInternals(e); i++)
         RecursiveSelectConnections(info, selected, GetGroupInternal(e, i),
                                    type, connection_list, num);
   else if (GetElementType(e) == NEURON_TYPE)
      /* Look if the connections from and to this neuron have to be selected */
      for (i = 0; i < GetNeuronNumConnections(e); i++) {
         connection = GetNeuronConnection(e, i);

         /* Get the other side of the connection */
         if (GetConnectionFrom(connection) == e)
            other = GetConnectionTo(connection);
         else
            other = GetConnectionFrom(connection);

         /* Check whether the other side of the connection is in the same
            hierarchy as this element. In that case the other side must be
            a child of "selected". We don't want to select connections inside
            a group */
         parent = GetElementParent(other);
         while ((parent != GetWindowInfoMainGroup(info)) &&
                (parent != selected))
            parent = GetElementParent(parent);
         if (parent != GetWindowInfoMainGroup(info))
            continue;  /* "other" is a child of "selected" */

         /* Handle each connectiontype */
         switch(type) {
            case INCOMING:
               /* If the connection is connecting to "e", add it */
               if (GetConnectionTo(connection) == e)
                  AddNewConnection(connection_list, num, connection);
               break;
            case OUTGOING:
               /* If the connection is connection from "e", add it */
               if (GetConnectionFrom(connection) == e)
                  AddNewConnection(connection_list, num, connection);
               break;
            case CONNECTING:
               /* Look if the other side of the connection is selected or
                  is a child of a selected group. In that case add
                  the connection */
               parent = other;
               while (!IsSelected(info, parent) &&
                      (parent != GetWindowInfoMainGroup(info)))
                  parent = GetElementParent(parent);
               if (IsSelected(info, parent))
                  AddNewConnection(connection_list, num, connection);
               break;
         }
      }
}


/* Routine to select connections.
   */
void SelectConnections(WindowInfo *info, SelectConnectionType type)
{
   Element **connection_list;
   int i, num;


   /* Initialize the connectionlist */
   connection_list = (Element **) malloc(sizeof(Element*));
   num = 0;

   /* Select connections using the recursive routine
      RecursiveSelectConnections() */
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      RecursiveSelectConnections(info, GetWindowInfoSelected(info, i),
                                 GetWindowInfoSelected(info, i), type,
                                 &connection_list, &num);

   /* Remove the selected items from the selected list and add the
      connections */
   SetWindowInfoSelected(info, NULL);
   for (i = 0; i < num; i++)
      AddWindowInfoSelected(info, connection_list[i]);

   /* Free the connectionlist */
   free(connection_list);
}


Element *GetConnectionEnd(WindowInfo *info, int x, int y)
{
   Element *e;
   Boolean b = True;
   int i;


   if ((e = GetElementWithinRange(info, x, y, NEURON_MASK, NO_MASK)))
      return(e);
   if ((e = GetElementWithinRange(info, x, y, GROUP_MASK, NO_MASK)) &&
       GetGroupOpened(e)) {
      /* find the "lowest" layer of groups */
      while (b) {
         b = False;
         for (i = GetGroupNumInternals(e) - 1; (i >= 0) && !b; i--)
            if ((GetElementType(GetGroupInternal(e, i)) == GROUP_TYPE) &&
                InElement(x, y, GetGroupInternal(e, i)) &&
                GetGroupOpened(GetGroupInternal(e, i))) {
               e = GetGroupInternal(e, i);
               b = True;
            }
      }
      return(e);
   }
   return(NULL);
}


/* Function to make an already opened group the current editlevel
   */
void EnterGroup(WindowInfo *info, Element *e)
{
   Element *edit = GetWindowInfoEditLevel(info);  /* the parent of e */
   int i;


   /* There is nothing selected anymore */
   NewSelected(info, NULL);

   /* Remove the group borders of the parent (if not the main group) and
      the element */
   if (edit != GetWindowInfoMainGroup(info))
      DrawElement(edit, True);
   DrawElement(e, True);

   /* Define the new editlevel */
   SetWindowInfoEditLevel(info, e);

   /* Redraw the siblings. Because there is a new editlevel, these elements
      have to be redrawn */
   for (i = 0; i < GetGroupNumInternals(edit); i++)
      if (GetGroupInternal(edit, i) != e)
         RecursiveDrawElement(GetGroupInternal(edit, i), False);

   /* Draw the new borders of the new editlevel and its parent */
   if (edit != GetWindowInfoMainGroup(info))
      DrawElement(edit, True);
   DrawElement(e, True);

   /* Set the default mode */
   SetDefaultMode(info);
}


/* Function to leave the current editlevel.
   The parent of the old editlevel will become the new editlevel
   */
void LeaveGroup(WindowInfo *info, Element *edit)
{
   Element *parent;
   int i;


   /* Leaving maingroup is impossible */
   if (GetWindowInfoMainGroup(info) == edit)
      return;

   /* Define the parent */
   parent = GetGroupParent(edit);

   /* There is nothing selected anymore */
   NewSelected(info, NULL);

   /* Remove the group borders of the parent (if not the main group) and
      the element */
   if (parent != GetWindowInfoMainGroup(info))
      DrawElement(parent, True);
   DrawElement(edit, True);

   /* Define the new editlevel */
   SetWindowInfoEditLevel(info, GetGroupParent(edit));

   /* Redraw the siblings. Because there is a new editlevel, these elements
      have to be redrawn */
   for (i = 0; i < GetGroupNumInternals(parent); i++)
      if (GetGroupInternal(parent, i) != edit)
         RecursiveDrawElement(GetGroupInternal(parent, i), False);

   /* Draw the new borders of the new and old editlevel */
   DrawElement(edit, True);
   DrawElement(parent, True);

   /* Set the default mode */
   SetDefaultMode(info);
}


/* Routine to cut selected items and their internals. The paste-list contains
   only the selected items (not their children) and the connections that
   are connecting them.
   */
static void _RecursiveCut(WindowInfo *info, Element *e)
{
   Element *c;
   int i;
   /* Remove the element from the list of elements */
   RemoveWindowInfoElement(info, e);

   switch(GetElementType(e)) {
      case GROUP_TYPE:
         /* Cut the internals */
         for (i = 0; i < GetGroupNumInternals(e); i++)
            _RecursiveCut(info, GetGroupInternal(e, i));
         break;
      case NEURON_TYPE:
         /* Remove the connections, add them to the paste-list and
            disconnect them */
         while (GetNeuronNumConnections(e)) {
            c = GetNeuronConnection(e, 0);
            RemoveWindowInfoElement(info, c);
            AddWindowInfoPaste(info, c);
            RemoveNeuronConnection(GetConnectionFrom(c), c);
            RemoveNeuronConnection(GetConnectionTo(c), c);
         }
         break;
      default:
         break;
   }
}


/* Routine to cut the selected items and put them in the paste-list. The
   selected items are put into the paste-list and the connections that are
   connecting them.
   */
void Cut(WindowInfo *info)
{
   Element *e;
   int i;


   /* Empty the paste-list */
   ClearWindowInfoPaste(info);
   for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
      e = GetWindowInfoSelected(info, i);

      /* Only non-connections are being cut (and connections connected to
         selected elements */
      if (GetElementType(e) != CONNECTION_TYPE) {
         /* Put an element in the paste-list, remove the element from the
            list of internals of the parent. _RecursiveCut() will remove
            the element from the list of elements in "info" and (if
            necessary) cut the children (group) or connections (neuron) */
         AddWindowInfoPaste(info, e);
         RemoveGroupInternal(GetElementParent(e), e);
         _RecursiveCut(info, e);
      }
   }

   /* Define the kind of paste-list, no selected elements anymore, redraw */
   SetWindowInfoPasteType(info, CUT);
   SetWindowInfoSelected(info, NULL);
   InterfaceReDraw(info);
}


/* Routine to copy selected elements.
   */
void Copy(WindowInfo *info)
{
   Element *e;
   int i;


   /* Clear the paste-list and all the selected elements to the paste-list */
   ClearWindowInfoPaste(info);
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      if (GetElementType(GetWindowInfoSelected(info, i)) != CONNECTION_TYPE)
         AddWindowInfoPaste(info, GetWindowInfoSelected(info, i));
   SetWindowInfoPasteType(info, COPY);
}


/* Routine to paste all the elements in the paste-list recursively after they
   have been cut.
   */
static void _RecursivePasteAfterCut(WindowInfo *info, Element *e)
{
   int i;


   /* Add the current element. Elements don't have to be added to the internals
      of the parent, because this has already been done in _PasteAfterCut()
      or they are children of the cutted elements and their internal structure
      is still intact */
   AddWindowInfoElement(info, e);
   switch(GetElementType(e)) {
      case CONNECTION_TYPE:
         /* Attach the connection to its from- and to-elements */
         AddNeuronConnection(GetConnectionFrom(e), e);
         AddNeuronConnection(GetConnectionTo(e), e);
         break;
      case GROUP_TYPE:
         /* Paste the internals */
         for (i = 0; i < GetGroupNumInternals(e); i++)
            _RecursivePasteAfterCut(info, GetGroupInternal(e, i));
         break;
      default:
         break;
   }
}


/* Routine to paste elements that have been cut. Move the elements over (dx,dy).
   */
static void _PasteAfterCut(WindowInfo *info, Element *edit, int dx, int dy)
{
   Element *e;
   int i;


   /* Move elements to new position */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++)
      if (GetElementType(GetWindowInfoPaste(info, i)) != CONNECTION_TYPE)
         RecursiveMoveElement(GetWindowInfoPaste(info, i), dx, dy);

   /* Now add the cutted elements to the current edit-lvl and call
      _RecursivePasteAfterCut() */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++) {
      e = GetWindowInfoPaste(info, i);
      if (GetElementType(e) != CONNECTION_TYPE) {
         AddGroupInternal(edit, e);
      }
      _RecursivePasteAfterCut(info, e);
   }

   /* Now the internal structure is correct, so we can define the
      positions of the connections. Meanwhile make every non-connection
      selected */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++) {
      if (GetElementType(GetWindowInfoPaste(info, i)) == CONNECTION_TYPE)
         SetConnectionPositions(GetWindowInfoPaste(info, i));
      else
         AddWindowInfoSelected(info, GetWindowInfoPaste(info, i));
   }

   /* Cutted elements can be pasted only once */
   ClearWindowInfoPaste(info);
}


/* This routine is the first routine called to paste copied elements
   recursively. It creates neurons and groups that are copied. Neurons are
   added to the CopyList "copy". This list is used to define the new ID's of
   the neurons from/to which the new connections have to be created.
   "dx" and "dy" is the difference between the mouse position and the
   top-left position of the elements that are pasted.
   */
static void _FirstRecursivePasteAfterCopy(WindowInfo *info, Element *parent,
                                          Element *e, int dx, int dy,
                                          CopyList *list)
{
   Element *new;
   Position new_x = GetElementXPosition(e) + dx;
   Position new_y = GetElementYPosition(e) + dy;
   int i;


   switch (GetElementType(e)) {
      case NEURON_TYPE:
         /* Create a new neuron, add it to the internals of its parent and
            add the neuron to the copylist */
         new = CreateNeuron(parent, new_x, new_y);
         AddGroupInternal(parent, new);
         AddCopy(list, e, new);
         break;
      case GROUP_TYPE:
         /* Crate a new group, add it to the internals of its parent and call
            this routine again to create its children */
         new = CreateGroup(parent, new_x, new_y, GetElementWidth(e),
                           GetElementHeight(e), GetGroupOpened(e),
                           GetGroupAbleOpening(e), GetGroupConnectionState(e));
         AddGroupInternal(parent, new);
         for (i = 0; i < GetGroupNumInternals(e); i++)
            _FirstRecursivePasteAfterCopy(info, new, GetGroupInternal(e, i),
                                          dx, dy, list);
         break;
      default:
         break;
   }
}



/* This routine is the second routine called to paste copied elements
   recursively. It recursively creates connections from and to elements
   that have been created by _FirstRecursivePasteAfterCopy(). If the
   element is a group, this routine will be called with its internals.
   */
static void _SecondRecursivePasteAfterCopy(WindowInfo *info, Element *e,
                                           CopyList *list)
{
   Element *c, *new;
   int i;


   switch(GetElementType(e)) {
      case NEURON_TYPE:
         for (i = 0; i < GetNeuronNumConnections(e); i++) {
            c = GetNeuronConnection(e, i);
            /* Only create a new connection if this connection hasn't been
               created yet */
            if (GetCopy(list, c) == c) {
               /* Create a new connection and set the positions */
               new = CreateConnection(GetCopy(list, GetConnectionFrom(c)),
                                      GetCopy(list, GetConnectionTo(c)),
                                      GetConnectionState(c));
               SetConnectionPositions(new);
```

```
                /* Add this connection to the copy-list, to avoid multiple
                   copies of one connection */
                AddCopy(list, c, new);
             }
          }
          break;
       case GROUP_TYPE:
          /* Copy the connections of the internals */
          for (i = 0; i < GetGroupNumInternals(e); i++)
             _SecondRecursivePasteAfterCopy(info, GetGroupInternal(e, i), list);
          break;
       default:
          break;
    }
}


/* Routine to paste the elements that were added in the paste-list to be
   copied. "dx" and "dy" is the difference between the current mouse-position
   and the top-left position of the elements that are being pasted.
   */
static void _PasteAfterCopy(WindowInfo *info, Element *edit, int dx, int dy)
{
   CopyList list;
   int i;


   /* Init the list of copies. No copies made yet */
   InitList(&list);

   /* Now make copies of all the elements but the connections */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++)
      _FirstRecursivePasteAfterCopy(info, edit, GetWindowInfoPaste(info, i),
                                    dx, dy, &list);

   /* Make connections between each copied element and made each copied
      element selected */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++) {
      _SecondRecursivePasteAfterCopy(info, GetWindowInfoPaste(info, i), &list);
      AddWindowInfoSelected(info, GetCopy(&list, GetWindowInfoPaste(info, i)));
   }

   /* Free the list of copies */
   FreeList(&list);
}


/* Routine used to paste elements. This is used to copy elements and to
   cut-paste elements.
   */
void Paste(WindowInfo *info)
{
   Element *edit = GetWindowInfoEditLevel(info);
   Position x = GetWindowInfoMenuXPosition(info);
   Position y = GetWindowInfoMenuYPosition(info);
   Position x1, y1, x2, y2;
   int i;


   /* If nothing to paste, skip */
   if (!GetWindowInfoNumPaste(info))
      return;

   /* Nothing selected .. yet */
   SetWindowInfoSelected(info, NULL);

   /* First define the top-left corner of the selected elements */
   for (i = 0; i < GetWindowInfoNumPaste(info); i++)
      if (GetElementType(GetWindowInfoPaste(info, i)) != CONNECTION_TYPE)
         ProcessPositionAndDimension(GetWindowInfoPaste(info, i));
   GetPositions(&x1, &y1, &x2, &y2);

   /* Call the routine to paste-after-cut or paste-after-copy. The elements
      have to be moved over (x-x1,y-y1), the difference between the current
      mouse position and the top-left corner of the elements in the
      paste-list */
   if (GetWindowInfoPasteType(info) == CUT)
      _PasteAfterCut(info, edit, x - x1, y - y1);
   else
      _PasteAfterCopy(info, edit, x - x1, y - y1);
   /* Propagate the sizes. Maybe the current edit-lvl has to become bigger
      because elements exceed the old group-borders */
   PropagateSizes(edit);

   /* Redraw */
   InterfaceReDraw(info);
}
```

6.8 Type

6.8.1 type.h

```
/*
* Header
*/
/*
* Log
*/

#ifndef TYPE_H
#define TYPE_H


#include <Xm/Xm.h>


typedef enum {
   EDIT_MODE, NEURON_MODE, CONNECTION_MODE } Mode;
#define NUM_MODES 3
typedef enum {
   ICON_BUTTON, DEICON_BUTTON,
   GROUP_BUTTON, UNGROUP_BUTTON, DELETE_BUTTON } CommandButton;
#define NUM_COMMAND_BUTTONS 5
typedef enum {
   COPY, CUT } PasteType;
typedef enum {
   VISIBLE, INVISIBLE, DEFAULT, PARENT } ConnectionState;
typedef enum {
   CONNECTION_TYPE, GROUP_TYPE, NEURON_TYPE,
   FUZZY_TYPE, CALCULATION_TYPE } ElementType;
typedef unsigned char ElementTypeMask;
#define CONNECTION_MASK (1 << CONNECTION_TYPE)
#define GROUP_MASK (1 << GROUP_TYPE)
#define NEURON_MASK (1 << NEURON_TYPE)
#define FUZZY_MASK (1 << FUZZY_TYPE)
#define CALCULATION_MASK (1 << CALCULATION_TYPE)
#define NO_MASK 0
#define ALL_MASK 63  /* 2^6 - 1 */
#define NUM_ELEMENT_TYPES 5


typedef struct element_str Element;
typedef struct win_info_str WindowInfo;


typedef struct {
   struct element_str *from, *to;
   Position x1, y1, x2, y2;
   ConnectionState state;
} ConnectionInfo;
```

```
typedef struct {
   Boolean opened, able_opening;
   struct element_str **internals;
   int num_internals;
   ConnectionState connection_state;
   Boolean main_group;
} GroupInfo;
typedef struct {
   struct element_str **connections;
   int num_connections;
} NeuronInfo;
struct element_str {
   int id;  /* TMP */
   ElementType type;
   Position x, y;
   Dimension width, height;
   void *parent;  /* not used for connections */
   union {
      ConnectionInfo connection;
      GroupInfo group;
      NeuronInfo neuron;
   } info;
};
struct win_info_str{
   Widget drawing_area, hori, verti;
   Widget buttons[NUM_MODES], selected_button;
   Widget command_buttons[NUM_COMMAND_BUTTONS];
   Element *main_element, *edit_level;
   Element **selected, **elements, **paste;
   int num_selected, num_elements, num_paste;
   PasteType paste_type;
   Position menu_x, menu_y;
   Mode mode, default_mode;
   float zoom;
   Position x_offset, y_offset;
   Boolean show_connection, show_id, show_direction;
};


/* Calls to create/change Elements */
#define SetElementXPosition(e, x_pos) ((e)->x = (x_pos))
#define SetElementYPosition(e, y_pos) (e->y = (y_pos))
#define SetElementWidth(e, w) ((e)->width = (w))
#define SetElementHeight(e, h) ((e)->height = (h))
#define SetElementParent(e, p) ((e)->parent = (p))
#define DestroyElement(e) (free(e))

/* Calls to examine Elements */
#define GetElementID(e) ((e)->id)
#define GetElementType(e) ((e)->type)
#define GetElementXPosition(e) ((e)->x)
#define GetElementYPosition(e) ((e)->y)
#define GetElementWidth(e) ((e)->width)
#define GetElementHeight(e) ((e)->height)
Element *GetElementParent(Element*);
WindowInfo *GetElementInfo(Element*);

/* Calls to create/change Connections */
Element *CreateConnection(Element*, Element*, ConnectionState);
#define SetConnectionFrom(e, f) ((e)->info.connection.from = (f))
#define SetConnectionTo(e, t) ((e)->info.connection.to = (t))
#define SetConnectionFromXPosition(e, x) ((e)->info.connection.x1 = (x))
#define SetConnectionFromYPosition(e, y) ((e)->info.connection.y1 = (y))
#define SetConnectionToXPosition(e, x) ((e)->info.connection.x2 = (x))
#define SetConnectionToYPosition(e, y) ((e)->info.connection.y2 = (y))
#define SetConnectionState(e, s) ((e)->info.connection.state = (s))

/* Calls to examine Connections */
#define GetConnectionFrom(e) ((e)->info.connection.from)
#define GetConnectionTo(e) ((e)->info.connection.to)
#define GetConnectionFromXPosition(e) ((e)->info.connection.x1)
```

```
#define GetConnectionFromYPosition(e) ((e)->info.connection.y1)
#define GetConnectionToXPosition(e) ((e)->info.connection.x2)
#define GetConnectionToYPosition(e) ((e)->info.connection.y2)
#define GetConnectionState(e) ((e)->info.connection.state)

/* Calls to create/change Groups */
Element *CreateGroup(Element*, Position, Position, Dimension, Dimension,
                     Boolean, Boolean, ConnectionState);
#define SetGroupOpened(e, b) ((e)->info.group.opened = (b))
#define SetGroupAbleOpening(e, b) ((e)->info.group.able_opening = (b))
Boolean SetGroupInternal(Element*, int, Element*);
Boolean AddGroupInternal(Element*, Element*);
Boolean RemoveGroupInternal(Element*, Element*);
#define SetGroupConnectionState(e, s) ((e)->info.group.connection_state = (s))

/* Calls to examine Groups */
#define GetGroupOpened(e) ((e)->info.group.opened)
#define GetGroupAbleOpening(e) ((e)->info.group.able_opening)
#define GetGroupInternal(e, i) \
   (((e)->info.group.num_internals <= (i)) ? \
   NULL : (e)->info.group.internals[(i)])
#define GetGroupNumInternals(e) ((e)->info.group.num_internals)
#define GetGroupConnectionState(e) ((e)->info.group.connection_state)
#define GetGroupParent(e) ((e)->info.group.main_group ? NULL : (e)->parent)
#define GetGroupMainGroup(e) ((e)->info.group.main_group)

/* Calls to create/change Neurons */
Element *CreateNeuron(Element*, Position, Position);
Boolean AddNeuronConnection(Element*, Element*);
void RemoveNeuronConnection(Element*, Element*);

/* Calls to examine Neurons */
#define GetNeuronConnection(e, i) \
   (((e)->info.neuron.num_connections <= (i)) ? \
   NULL : (e)->info.neuron.connections[(i)])
#define GetNeuronNumConnections(e) ((e)->info.neuron.num_connections)
#define GetNeuronParent(e) ((e)->parent)

/* Calls to create/change WindowInfo */
WindowInfo *CreateWindowInfo(Mode, float, Boolean, Boolean);
#define SetWindowInfoDrawingArea(i, w) ((i)->drawing_area = (w))
#define SetWindowInfoHorizontalScrollBar(i, w) ((i)->hori = (w))
#define SetWindowInfoVerticalScrollBar(i, w) ((i)->verti = (w))
#define SetWindowInfoButton(i, m, w) ((i)->buttons[(m)] = (w))
#define SetWindowInfoSelectedButton(i, w) ((i)->selected_button = (w))
#define SetWindowInfoCommandButton(i, b, w) ((i)->command_buttons[(b)] = (w))
#define SetWindowInfoEditLevel(i, e) ((i)->edit_level = (e))
void SetWindowInfoSelected(WindowInfo*, Element*);
Boolean AddWindowInfoSelected(WindowInfo*, Element*);
Boolean SetWindowInfoElement(WindowInfo*, int, Element*);
Boolean RemoveWindowInfoElement(WindowInfo*, Element*);
#define AddWindowInfoElement(i, e) \
   (SetWindowInfoElement((i), (i)->num_elements, (e)))
void ClearWindowInfoPaste(WindowInfo*);
Boolean AddWindowInfoPaste(WindowInfo*, Element*);
#define SetWindowInfoPasteType(i, t) ((i)->paste_type = (t))
#define SetWindowInfoMenuXPosition(i, x) ((i)->menu_x = (x))
#define SetWindowInfoMenuYPosition(i, y) ((i)->menu_y = (y))
#define SetWindowInfoMode(i, m) ((i)->mode = (m))
#define SetWindowInfoDefaultMode(i, m) ((i)->default_mode = (m))
#define SetWindowInfoZoom(i, z) ((i)->zoom = (z))
#define SetWindowInfoXOffset(i, x) ((i)->x_offset = (x))
#define SetWindowInfoYOffset(i, y) ((i)->y_offset = (y))
#define SetWindowInfoShowConnection(i, b) ((i)->show_connection = (b))
#define SetWindowInfoShowID(i, b) ((i)->show_id = (b))
#define SetWindowInfoShowDirection(i, b) ((i)->show_direction = (b))

/* Calls to examine WindowInfo */
#define GetWindowInfoDrawingArea(i) ((i)->drawing_area)
#define GetWindowInfoHorizontalScrollBar(i) ((i)->hori)
#define GetWindowInfoVerticalScrollBar(i) ((i)->verti)
```

```
#define GetWindowInfoButton(i, m) ((i)->buttons[(m)])
#define GetWindowInfoSelectedButton(i) ((i)->selected_button)
#define GetWindowInfoCommandButton(i, b) ((i)->command_buttons[(b)])
#define GetWindowInfoMainGroup(i) ((i)->main_element)
#define GetWindowInfoEditLevel(i) ((i)->edit_level)
#define GetWindowInfoSelected(i, idx) \
   (((i)->num_selected <= (idx)) ? NULL : (i)->selected[(idx)])
#define GetWindowInfoNumSelected(i) ((i)->num_selected)
#define GetWindowInfoElement(i, idx) \
   (((i)->num_elements <= (idx)) ? NULL : (i)->elements[(idx)])
#define GetWindowInfoNumElements(i) ((i)->num_elements)
#define GetWindowInfoPaste(i, idx) ((i)->paste[(idx)])
#define GetWindowInfoNumPaste(i) ((i)->num_paste)
#define GetWindowInfoPasteType(i) ((i)->paste_type)
#define GetWindowInfoMenuXPosition(i) ((i)->menu_x)
#define GetWindowInfoMenuYPosition(i) ((i)->menu_y)
#define GetWindowInfoMode(i) ((i)->mode)
#define GetWindowInfoDefaultMode(i) ((i)->default_mode)
#define GetWindowInfoZoom(i) ((i)->zoom)
#define GetWindowInfoXOffset(i) ((i)->x_offset)
#define GetWindowInfoYOffset(i) ((i)->y_offset)
#define GetWindowInfoShowConnection(i) ((i)->show_connection)
#define GetWindowInfoShowID(i) ((i)->show_id)
#define GetWindowInfoShowDirection(i) ((i)->show_direction)

#endif
```

6.8.2 type.c

```
/*
* Header
*/
/*
* Log
*/


#include <macros.h>
/*#include <malloc.h>*/
#include "type.h"
#include "constants.h"


/*******************************
   *
   * Static calls
   *
   *******************************/


/* Function to create an element. this function is used for all types of
   elements
   */
static Element *CreateElement(ElementType type, Position x, Position y,
                              Dimension width, Dimension height, Element *from,
                              Element *to, ConnectionState state,
                              Boolean opened, Boolean able_opening,
                              Boolean main_group, void *parent)
{
   Element *e;
   static int num = 0;


   if (!(e = (Element *) malloc(sizeof(Element))))
      return(NULL);

   e->id = num;
   num++;
   e->type = type;
   e->x = x;
   e->y = y;
   e->width = width;
   e->height = height;
   e->parent = parent;
   switch(type) {
      case CONNECTION_TYPE:
         e->info.connection.from = from;
         e->info.connection.to = to;
         e->info.connection.state = state;
         if (!AddNeuronConnection(from, e) || !AddNeuronConnection(to, e))
            return(NULL);
         break;
      case GROUP_TYPE:
         e->info.group.opened = opened;
         e->info.group.able_opening = able_opening;
         e->info.group.internals = NULL;
         e->info.group.num_internals = 0;
         e->info.group.connection_state = state;
         e->info.group.main_group = main_group;
         break;
      case NEURON_TYPE:
         e->info.neuron.connections = NULL;
         e->info.neuron.num_connections = 0;
         break;
      case FUZZY_TYPE:
      case CALCULATION_TYPE:
         break;
   }

   /* Push Element in list */
   AddWindowInfoElement(GetElementInfo(e), e);

   return(e);
}


/******************************
   *
   * Calls to examine elements
   *
   ******************************/


Element *GetElementParent(Element *element)
{
   if (element->type == CONNECTION_TYPE)
      return(GetElementParent(element->info.connection.from));
   else
      return(element->parent);
}


WindowInfo *GetElementInfo(Element *element)
{
   if (element->type == CONNECTION_TYPE)
      return(GetElementInfo(element->info.connection.from));
   else if ((element->type == GROUP_TYPE) && element->info.group.main_group)
      return((WindowInfo *) element->parent);
   else
      return(GetElementInfo(element->parent));
}


/******************************
   *
   * Calls to create/change connections
   *
   ******************************/


Element *CreateConnection(Element *from, Element *to, ConnectionState state)
{
   Position x, y;
   Dimension width, height;


   /* x, y, width and height only used for connections to itself */
   x = from->x;
   y = from->y - NEURON_WIDTH;
   width = NEURON_WIDTH;
   height = NEURON_WIDTH;

   return(CreateElement(CONNECTION_TYPE, x, y, width, height, from, to, state,
                        FALSE, FALSE, FALSE, (void *) from));
}


/*******************************
   *
   * Calls to create/change groups
   *
   *******************************/


Element *CreateGroup(Element *parent, Position x, Position y, Dimension w,
                     Dimension h, Boolean opened, Boolean able_opening,
                     ConnectionState state)
{
   return(CreateElement(GROUP_TYPE, x, y, w, h, NULL, NULL, state, opened,
                        able_opening, FALSE, (void *) parent));
}


Boolean SetGroupInternal(Element *parent, int i, Element *internal)
{
   if ((i < 0) || (i > parent->info.group.num_internals))
      return(False);

   if (i == parent->info.group.num_internals) {
      if (!(parent->info.group.internals =
            (Element **) realloc(parent->info.group.internals,
                                 (i + 1) * sizeof(Element *))))
         return(False);
      parent->info.group.num_internals++;
   }

   parent->info.group.internals[i] = internal;
   return(True);
}


Boolean AddGroupInternal(Element *parent, Element *internal)
{
   if (!SetGroupInternal(parent, parent->info.group.num_internals, internal))
      return(False);
   internal->parent = (void *) parent;
   return(True);
}


Boolean RemoveGroupInternal(Element *e, Element *internal)
{
   int i, j;


   for (i = 0; (i < e->info.group.num_internals) &&
        (e->info.group.internals[i] != internal); i++) ;
   if (i == e->info.group.num_internals)
      return(FALSE);
   for (j = i; j < e->info.group.num_internals - 1; j++)
      e->info.group.internals[j] = e->info.group.internals[j + 1];
   e->info.group.num_internals--;
   return(TRUE);
}


/******************************
   *
   * Calls to create/change neurons
   *
   ******************************/


Element *CreateNeuron(Element *parent, Position x, Position y)
{
   return(CreateElement(NEURON_TYPE, x, y, 0, 0, NULL, NULL, DEFAULT, FALSE,
                        FALSE, FALSE, (void *) parent));
}


Boolean AddNeuronConnection(Element *e, Element *c)
{
   if (!(e->info.neuron.connections =
         (Element **) realloc(e->info.neuron.connections,
                              (e->info.neuron.num_connections + 1) *
                              sizeof(Element*))))
      return(FALSE);
   e->info.neuron.connections[e->info.neuron.num_connections] = c;
   e->info.neuron.num_connections++;
   return(TRUE);
}


void RemoveNeuronConnection(Element *e, Element *connection)
{
   int i, j;


   for (i = 0; (i < e->info.neuron.num_connections) &&
        (e->info.neuron.connections[i] != connection); i++) ;

   if (i == e->info.neuron.num_connections)
      return;
   for (j = i; j < e->info.neuron.num_connections - 1; j++)
      e->info.neuron.connections[j] = e->info.neuron.connections[j + 1];
   e->info.neuron.num_connections--;
}


/******************************
   *
   * Calls to create/change windowinfo
   *
   ******************************/


WindowInfo *CreateWindowInfo(Mode mode, float zoom, Boolean show_connection,
                             Boolean show_id)
{
   WindowInfo *info;


   if (!(info = (WindowInfo *) malloc(sizeof(WindowInfo))))
      return(NULL);
   info->selected_button = NULL;
   info->selected = (Element **) malloc(sizeof(Element *));
   info->num_selected = 0;
   info->elements = NULL;
   info->num_elements = 0;
   info->paste = NULL;
   info->num_paste = 0;
   info->mode = mode;
   info->default_mode = mode;
   info->zoom = zoom;
   info->show_connection = show_connection;
   info->show_id = show_id;
   info->main_element = CreateElement(GROUP_TYPE, 0, 0, 0, 0, NULL, NULL,
                                 DEFAULT, TRUE, TRUE, TRUE, (void *) info);
   info->edit_level = info->main_element;
   return(info);
}


void SetWindowInfoSelected(WindowInfo *info, Element *element)
{
   if (element) {
      info->selected[0] = element;
      info->num_selected = 1;
   }
   else
      info->num_selected = 0;
}


Boolean AddWindowInfoSelected(WindowInfo *info, Element *element)
{
   int i;


   if (element) {
      for (i = 0; (i < info->num_selected) &&
           (info->selected[i] != element); i++) ;
      if (i != info->num_selected)
         return(False);
      if (!(info->selected = (Element **)realloc(info->selected,
                                         (info->num_selected + 1) *
                                         sizeof(Element *))))
         return(False);
      info->selected[info->num_selected] = element;
      info->num_selected++;
   }
   return(True);
}


Boolean SetWindowInfoElement(WindowInfo *info, int i, Element *element)
{
   if ((i < 0) || (i > info->num_elements))
      return (False);

   if (i == info->num_elements) {
      if (!(info->elements = (Element **) realloc(info->elements,
                                          (i + 1) * sizeof(Element *))))
         return(False);
      info->num_elements++;
   }

   info->elements[i] = element;
   return(True);
}


Boolean RemoveWindowInfoElement(WindowInfo* info, Element *element)
{
   int i, j;


   for (i = 0; (i < info->num_elements) &&
         (info->elements[i] != element); i++) ;
   if (i == info->num_elements)
      return(False);

   for (j = i; j < info->num_elements - 1; j++)
      info->elements[j] = info->elements[j + 1];
   info->num_elements--;
   return(True);
}


void ClearWindowInfoPaste(WindowInfo *info)
{
   if (info->num_paste)
      free(info->paste);
   info->num_paste = 0;
}


Boolean AddWindowInfoPaste(WindowInfo *info, Element *element)
{
   if (!(info->paste = (Element **) realloc(info->paste,
                                    (info->num_paste + 1) *
                                    sizeof(Element*))))
      return(False);

   info->paste[info->num_paste] = element;
   info->num_paste++;
   return(True);
}
```

6.9 Visualize

6.9.1 visualize.h

```
/*
* Header
*/
/*
* Log
*/


#ifndef VISUALIZE_H
#define VISUALIZE_H


#include "constants.h"
#include "operations.h"


void ClassInitialize();
void DrawElement(Element*, Boolean);
void RecursiveDrawElement(Element*, Boolean);
void DrawCorners(WindowInfo*);
void DrawSelected(WindowInfo*);
void DrawLine(Widget, Position, Position, Position, Position);
void DrawArea(Widget, Position, Position, Position, Position);
void DrawGroup(Widget, Position, Position, Position, Position);
void DrawBorders(Element*);
void ReDraw(WindowInfo*);

#endif
```

6.9.2 visualize.c

```
#include <math.h>
#include <stdio.h>   /* TMP */
#include <Ixm.h>
#include "interface.h"
#include "operations.h"
#include "visualize.h"


/* Colors */
#define GROUP_COLOR IxmColorDarkYellow
```

```
#define OPENED_GROUP_COLOR IxmColorDarkYellow
#define NEURON_COLOR IxmColorDarkGreen
#define CONNECTION_COLOR IxmColorLightGray
#define EDIT_LEVEL_COLOR IxmColorOrange
#define SELECTED_COLOR IxmColorWhite
#define EDIT_GROUP_COLOR IxmColorYellow
#define EDIT_OPENED_GROUP_COLOR IxmColorYellow
#define EDIT_NEURON_COLOR IxmColorGreen
/* Font */
#define ID_FONT IxmFontSHelvB12
/* Line width */
#define OPENED_GROUP_LINE_WIDTH 1
#define EDIT_LEVEL_LINE_WIDTH 3
#define NEURON_CONNECTION_LINE_WIDTH 1
#define GROUP_CONNECTION_LINE_WIDTH 3
/* Line style */
#define OPENED_GROUP_LINE_STYLE LineSolid
#define EDIT_LEVEL_LINE_STYLE LineSolid
/* Width/spacing */
#define REDRAW_INTERVAL 50              /* interval before redrawing selected */
#define FRONT_LOCATION (2.0/3.0)     /* position of indicator on a connection */
#define PERPEND_DIST 8  /* distance between front indicator and perpendicular */
#define SIDE_DIST 4      /* distance between connection and side of indicator */
#define SELECTED_CONNECTION_PADDING 8      /* distance between connection and */
                                                         /* selecting circles */
#define BORDER_PADDING 5              /* distance between element and borders */
#define BORDER_LEN 6                           /* length of border indicators */
/* Indexes in GC arrays */
#define GROUP_CONNECTION_TYPE (NUM_ELEMENT_TYPES)
#define OPENED_GROUP_TYPE (NUM_ELEMENT_TYPES + 1)
#define EDIT_LEVEL_TYPE (NUM_ELEMENT_TYPES + 2)
#define SELECTED_TYPE (NUM_ELEMENT_TYPES + 3)
#define NUM_GCS (NUM_ELEMENT_TYPES + 4)

/* Utility for creating two graphics contexts: a normal one and one
   using the XOR as drawing-function */
#define DefineGC(g, type, mask) \
   (g)[0][type] = XCreateGC(Ixm_display, DefaultRootWindow(Ixm_display), \
   mask, &gcv); \
   (g)[1][type] = XCreateGC(Ixm_display, DefaultRootWindow(Ixm_display), \
   mask | GCFunction, &gcv)

/* Utility for drawing the selected items */
#define DrawCircle(x, y) \
   InterfaceIxmDrawArc(w, gc[1][SELECTED_TYPE], x - SELECTED_WIDTH / 2, \
   y - SELECTED_WIDTH / 2, SELECTED_WIDTH, SELECTED_WIDTH, 0, 64 * 360)
#define DrawBorder(x1, y1, x2, y2, x3, y3) \
   InterfaceIxmDrawLine(w, gc[1][SELECTED_TYPE], x1, y1, x2, y2); \
   InterfaceIxmDrawLine(w, gc[1][SELECTED_TYPE], x1, y1, x3, y3)
/* Utility to round doubles and convert to ints */
#define my_round(x) ((int) (floor((x)+0.5)))


typedef struct {
   Element *from, *to;
} DrawnConnection;
typedef enum {
   ARROW, LINE_AND_ARROW
} DrawConnectionType;

static GC gc[2][NUM_GCS];  /* Normal GCs. Two seperate for xor/no xor */
static GC edit_gc[2][NUM_GCS];  /* GCs for elements in current edit lvl */


static unsigned long GetPixelValue(char *name)  /* TMP! */
{
   Colormap cmap = DefaultColormap(Ixm_display, DefaultScreen(Ixm_display));
   XColor color;
   char name2[50];
   strcpy(name2, name);
   XParseColor(Ixm_display, cmap, name2, &color);
   XAllocColor(Ixm_display, cmap, &color);
   return(color.pixel);
}


static void DrawArrow(Widget w, GC gc, int x1, int y1, int x2, int y2)
{
   double delta, a, b, c, d;


   /* Only draw arrows if distance != 0 */
   if (!((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)))
      return;


   delta = 1.0 / sqrt((x2 - x1) * (x2 - x1) +
                      (y2 - y1) * (y2 - y1));
   a = 1.0 - FRONT_LOCATION;
   b = 1.0 - FRONT_LOCATION + delta * PERPEND_DIST;
   c = FRONT_LOCATION - delta * PERPEND_DIST;
   d = delta * SIDE_DIST;
   InterfaceIxmDrawLine(w, gc, my_round(a * x1 + FRONT_LOCATION * x2),
                        my_round(a * y1 + FRONT_LOCATION * y2),
                        my_round(b * x1 + c * x2 - d * y2 + d * y1),
                        my_round(b * y1 + c * y2 - d * x1 + d * x2));
   InterfaceIxmDrawLine(w, gc, my_round(a * x1 + FRONT_LOCATION * x2),
                        my_round(a * y1 + FRONT_LOCATION * y2),
                        my_round(b * x1 + c * x2 + d * y2 - d * y1),
                        my_round(b * y1 + c * y2 + d * x1 - d * x2));
}


static void DrawConnection(Widget w, GC gc, Element *e,
                           DrawConnectionType type)
{
   Element *from, *to;


   if (!ShowConnection(e))
      return;

   from = ClosedParent(GetConnectionFrom(e));
   to = ClosedParent(GetConnectionTo(e));

   /* If it is a connection inside a iconified group, leave */
   if ((from == to) && !IsVisible(GetConnectionFrom(e)))
      return;

   if (type == LINE_AND_ARROW)
      if (from == to)
         InterfaceIxmDrawArc(w, gc, LeftPosition(from), TopPosition(from) -
                             ExternalHeight(from), ExternalWidth(from),
                             ExternalHeight(from), 0, 64 * 360);
      else
         InterfaceIxmDrawLine(w, gc, GetConnectionFromXPosition(e),
                              GetConnectionFromYPosition(e),
                              GetConnectionToXPosition(e),
                              GetConnectionToYPosition(e));


   /* Now draw the arrow */
   if (from == to)
      DrawArrow(w, gc, RightPosition(from), TopPosition(from) -
                ExternalHeight(from), LeftPosition(from), TopPosition(from) -
                ExternalHeight(from));
   else
      DrawArrow(w, gc, GetConnectionFromXPosition(e),
                GetConnectionFromYPosition(e),
                GetConnectionToXPosition(e), GetConnectionToYPosition(e));
}


void ClassInitialize()
{
   XGCValues gcv;


   gcv.function = GXxor;
   gcv.background = IxmColorBlack;
   gcv.font = ID_FONT->fid;

   /* Define all GCs */
   gcv.foreground = CONNECTION_COLOR;
   gcv.line_width = NEURON_CONNECTION_LINE_WIDTH;
   DefineGC(gc, CONNECTION_TYPE, GCForeground | GCLineWidth | GCFont);
   DefineGC(edit_gc, CONNECTION_TYPE, GCForeground | GCLineWidth | GCFont);
   gcv.line_width = GROUP_CONNECTION_LINE_WIDTH;
   DefineGC(gc, GROUP_CONNECTION_TYPE, GCForeground | GCLineWidth | GCFont);
   DefineGC(edit_gc, GROUP_CONNECTION_TYPE, GCForeground |
            GCLineWidth | GCFont);
   gcv.foreground = GROUP_COLOR;
   DefineGC(gc, GROUP_TYPE, GCForeground | GCFont);
   DefineGC(gc, FUZZY_TYPE, GCForeground | GCFont);
   DefineGC(gc, CALCULATION_TYPE, GCForeground | GCFont);
   gcv.foreground = EDIT_GROUP_COLOR;
   DefineGC(edit_gc, GROUP_TYPE, GCForeground | GCFont);
   DefineGC(edit_gc, FUZZY_TYPE, GCForeground | GCFont);
   DefineGC(edit_gc, CALCULATION_TYPE, GCForeground | GCFont);
   gcv.foreground = NEURON_COLOR;
   DefineGC(gc, NEURON_TYPE, GCForeground | GCFont);
   gcv.foreground = EDIT_NEURON_COLOR;
   DefineGC(edit_gc, NEURON_TYPE, GCForeground | GCFont);
   gcv.foreground = OPENED_GROUP_COLOR;
   gcv.line_width = OPENED_GROUP_LINE_WIDTH;
   gcv.line_style = OPENED_GROUP_LINE_STYLE;
   DefineGC(gc, OPENED_GROUP_TYPE, GCForeground | GCLineWidth |
            GCLineStyle | GCFont);
   gcv.foreground = EDIT_OPENED_GROUP_COLOR;
   DefineGC(edit_gc, OPENED_GROUP_TYPE, GCForeground | GCLineWidth |
            GCLineStyle | GCFont);
   gcv.foreground = EDIT_LEVEL_COLOR;
   gcv.line_width = EDIT_LEVEL_LINE_WIDTH;
   gcv.line_style = EDIT_LEVEL_LINE_STYLE;
   DefineGC(gc, EDIT_LEVEL_TYPE, GCForeground | GCLineWidth |
            GCLineStyle | GCFont);
   DefineGC(edit_gc, EDIT_LEVEL_TYPE, GCForeground | GCLineWidth |
            GCLineStyle | GCFont);
   gcv.foreground = SELECTED_COLOR;
   DefineGC(gc, SELECTED_TYPE, GCForeground | GCFont);
   DefineGC(edit_gc, SELECTED_TYPE, GCForeground | GCFont);
}


void DrawElement(Element *e, Boolean use_xor)
{
   WindowInfo *info = GetElementInfo(e);
   Widget w = GetWindowInfoDrawingArea(info);
   GC use_gc;
   Element *from, *to, *parent;
   Boolean in_edit_lvl = False;
   char str[10];
   int id_x, id_y, str_len;


   if (!e)
      return;

   /* Only draw if not the main group */
   if ((GetElementType(e) == GROUP_TYPE) &&
     (GetWindowInfoMainGroup(info) == e))
    return;

 /* Check whether element is inside the edit level */
 if (GetElementType(e) != CONNECTION_TYPE) {
    parent = e;
    while ((parent != GetWindowInfoMainGroup(info)) &&
           (parent != GetWindowInfoEditLevel(info)))
       parent = GetElementParent(parent);
    if (parent == GetWindowInfoEditLevel(info))
          in_edit_lvl = True;
 }

 use_gc = in_edit_lvl ? edit_gc[use_xor][GetElementType(e)] :
    gc[use_xor][GetElementType(e)];

 /* Draw the element */
 switch(GetElementType(e)) {
    case GROUP_TYPE:
       if (GetGroupOpened(e))
          if (e == GetWindowInfoEditLevel(info))
             InterfaceIxmDrawRectangle(w, gc[use_xor][EDIT_LEVEL_TYPE],
                                       LeftPosition(e), TopPosition(e),
                                       ExternalWidth(e), ExternalHeight(e));
          else
             InterfaceIxmDrawRectangle(w, in_edit_lvl ?
                                       edit_gc[use_xor][OPENED_GROUP_TYPE] :
                                       gc[use_xor][OPENED_GROUP_TYPE],
                                       LeftPosition(e), TopPosition(e),
                                       ExternalWidth(e), ExternalHeight(e));
       else
          InterfaceIxmFillRectangle(w, use_gc,
                                    LeftPosition(e), TopPosition(e),
                                    ExternalWidth(e), ExternalHeight(e));
       break;
    case FUZZY_TYPE:
    case CALCULATION_TYPE:
       InterfaceIxmFillRectangle(w, use_gc, LeftPosition(e),
                                 TopPosition(e), ExternalWidth(e),
                                 ExternalHeight(e));
       break;
    case NEURON_TYPE:
       InterfaceIxmFillArc(w, use_gc, LeftPosition(e), TopPosition(e),
                           NEURON_WIDTH, NEURON_WIDTH, 0, 64 * 360);
       break;
    case CONNECTION_TYPE:
       from = ClosedParent(GetConnectionFrom(e));
       to = ClosedParent(GetConnectionTo(e));
       if ((from != GetConnectionFrom(e)) ||
           (to != GetConnectionTo(e)))
          DrawConnection(w, gc[use_xor][GROUP_CONNECTION_TYPE], e,
                         LINE_AND_ARROW);
       else
          DrawConnection(w, use_gc, e, LINE_AND_ARROW);
       break;
 }


 /* Does the ID have to be drawn? */
 if (!GetWindowInfoShowID(info))
    return;

 /* Determine position of ID */
 sprintf(str, "%d", GetElementID(e));
 str_len = strlen(str);
 id_x = GetElementXPosition(e) - XTextWidth(ID_FONT, str, str_len) / 2;
 id_y = GetElementYPosition(e) + (ID_FONT->ascent - ID_FONT->descent) / 2;

 /* Determine the GC to draw the ID */
 use_gc = in_edit_lvl ? edit_gc[1][GetElementType(e)] :
    gc[1][GetElementType(e)];
```

```
   /* Draw the ID */
   switch(GetElementType(e)) {
      case GROUP_TYPE:
         if (!GetGroupOpened(e))
            InterfaceIxmDrawString(w, use_gc, id_x, id_y + GROUP_HEIGHT / 4,
                                   str);
         break;
      case FUZZY_TYPE:
      case CALCULATION_TYPE:
         InterfaceIxmDrawString(w, use_gc, id_x, id_y + GROUP_HEIGHT / 4, str);
         break;
      case NEURON_TYPE:
         InterfaceIxmDrawString(w, use_gc, id_x, id_y, str);
         break;
      case CONNECTION_TYPE:
         /* No connection ID's */
         break;
   }
}


static void _RecursiveDrawElement(Element *e, Boolean use_xor,
                                  Boolean closed_group, DrawnConnection **dc,
                                  int *num_drawn)
{
   GC use_gc;
   Element *from, *to;
   int i, j;


   if (!e)
      return;

   /* Draw element if this level is opened */
   if (!closed_group)
      DrawElement(e, use_xor);

   /* If the element type is group, draw internals */
   if (GetElementType(e) == GROUP_TYPE)
      for (i = 0; i < GetGroupNumInternals(e); i++)
         _RecursiveDrawElement(GetGroupInternal(e, i), use_xor,
                               closed_group || !GetGroupOpened(e), dc,
                               num_drawn);

   if (GetElementType(e) == NEURON_TYPE)
      for (i = 0; i < GetNeuronNumConnections(e); i++) {
         from = ClosedParent(GetConnectionFrom(GetNeuronConnection(e, i)));
         to = ClosedParent(GetConnectionTo(GetNeuronConnection(e, i)));

         /* Get GC */
         if ((from != GetConnectionFrom(GetNeuronConnection(e, i))) ||
             (to != GetConnectionTo(GetNeuronConnection(e, i))))
            use_gc = gc[use_xor][GROUP_CONNECTION_TYPE];
         else
            use_gc = gc[use_xor][CONNECTION_TYPE];

         /* Try to find an already drawn connection */
         for (j = 0; (j < *num_drawn) &&
              !(((*dc)[j].from == from) && ((*dc)[j].to == to)); j++) ;
         if (j != *num_drawn)
            /* Connection already drawn */
            continue;

         /* Try to find an "opposite" connection */
         for (j = 0; (j < *num_drawn) &&
              !(((*dc)[j].from == to) && ((*dc)[j].to == from)); j++) ;

         DrawConnection(GetWindowInfoDrawingArea(GetElementInfo(e)), use_gc,
                        GetNeuronConnection(e, i),
                        (j == *num_drawn) ? LINE_AND_ARROW : ARROW);
      (*dc) = (DrawnConnection *) realloc(*dc, (*num_drawn + 1) *
                                          sizeof(DrawnConnection));
      (*dc)[*num_drawn].from = from;
      (*dc)[*num_drawn].to = to;
      (*num_drawn)++;
    }
}


void RecursiveDrawElement(Element *e, Boolean use_xor)
{
  DrawnConnection *c = NULL;
  int num_drawn = 0;


  _RecursiveDrawElement(e, use_xor, False, &c, &num_drawn);
  free(c);
}


static void BlinkElementCallback(XtPointer client, XtIntervalId *id)
{
  DrawElement((Element *) client, True);
}


static void BlinkElement(Element *e, XtAppContext app)
{
  int i;


  if ((GetElementType(e) == GROUP_TYPE) && GetGroupOpened(e))
    for (i = 0; i < GetGroupNumInternals(e); i++)
      BlinkElement(GetGroupInternal(e, i), app);
  else {
    DrawElement(e, True);
    XtAppAddTimeOut(app, REDRAW_INTERVAL, BlinkElementCallback,
                    (XtPointer) e);
  }
}


void DrawCorners(WindowInfo *info)
{
  Element *e;
  Widget w;
  int i;


  w = GetWindowInfoDrawingArea(info);
  for (i = 0; i < GetWindowInfoNumSelected(info); i++) {
    e = GetWindowInfoSelected(info, i);
    if ((GetElementType(e) == CONNECTION_TYPE) &&
        (GetConnectionFrom(e) != GetConnectionTo(e))) {
      int x1 = GetConnectionFromXPosition(e);
      int y1 = GetConnectionFromYPosition(e);
      int x2 = GetConnectionToXPosition(e);
      int y2 = GetConnectionToYPosition(e);
      double delta;

      if (pow(x2 - x1, 2) || pow(y2 - y1, 2)) {
        delta = 1.0 / sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
        DrawCircle((int) (x1 + SELECTED_CONNECTION_PADDING *
                          delta * (y2 - y1)),
                   (int) (y1 + SELECTED_CONNECTION_PADDING *
                          delta * (x1 - x2)));
        DrawCircle((int) (x1 - SELECTED_CONNECTION_PADDING *
                          delta * (y2 - y1)),
                   (int) (y1 - SELECTED_CONNECTION_PADDING *
                          delta * (x1 - x2)));
        DrawCircle((int) (x2 + SELECTED_CONNECTION_PADDING *
                              delta * (y2 - y1)),
                        (int) (y2 + SELECTED_CONNECTION_PADDING *
                              delta * (x1 - x2)));
            DrawCircle((int) (x2 - SELECTED_CONNECTION_PADDING *
                              delta * (y2 - y1)),
                        (int) (y2 - SELECTED_CONNECTION_PADDING *
                              delta * (x1 - x2)));
         }
      }
      else {
         Position left, right, top, bottom;

         left = LeftPosition(e) - SELECTED_PADDING;
         right = RightPosition(e) + SELECTED_PADDING;
         top = TopPosition(e) - SELECTED_PADDING;
         bottom = BottomPosition(e) + SELECTED_PADDING;
         DrawCircle(left, top);
         DrawCircle(left, bottom);
         DrawCircle(right, top);
         DrawCircle(right, bottom);
      }
   }
}


void DrawSelected(WindowInfo *info)
{
   Widget w = GetWindowInfoDrawingArea(info);
   int i;


   if (!GetWindowInfoNumSelected(info))
      return;
   DrawCorners(info);
   for (i = 0; i < GetWindowInfoNumSelected(info); i++)
      BlinkElement(GetWindowInfoSelected(info, i),
                   XtWidgetToApplicationContext(w));
}


void DrawLine(Widget w, Position x1, Position y1, Position x2, Position y2)
{
   InterfaceIxmDrawLine(w, gc[1][CONNECTION_TYPE], x1, y1, x2, y2);
}


void DrawArea(Widget w, Position x1, Position y1, Position x2, Position y2)
{
   InterfaceIxmDrawRectangle(w, gc[1][SELECTED_TYPE], x1, y1,
                             x2 - x1, y2 - y1);
}


void DrawGroup(Widget w, Position x1, Position y1, Position x2, Position y2)
{
   /* Always use XOR and always use GC of editlevel, because groups can
      only be resized in current editlevel */
   InterfaceIxmDrawRectangle(w, edit_gc[1][OPENED_GROUP_TYPE], x1, y1,
                             x2 - x1, y2 - y1);
}


void DrawBorders(Element *e)
{
   Widget w;
   Position left, right, top, bottom;


   if (!e)
      return;
   w = GetWindowInfoDrawingArea(GetElementInfo(e));
   left = LeftPosition(e) - BORDER_PADDING;
   right = RightPosition(e) + BORDER_PADDING;
   top = TopPosition(e) - BORDER_PADDING;
   bottom = BottomPosition(e) + BORDER_PADDING;
   DrawBorder(left, top, left + BORDER_LEN, top, left, top + BORDER_LEN);
   DrawBorder(right, top, right - BORDER_LEN, top, right, top + BORDER_LEN);
   DrawBorder(left, bottom, left + BORDER_LEN, bottom, left,
              bottom - BORDER_LEN);
   DrawBorder(right, bottom, right - BORDER_LEN, bottom,
              right, bottom - BORDER_LEN);
}


void ReDraw(WindowInfo *info)
{
   fprintf(stderr, "name: %s\n", XtName(GetWindowInfoDrawingArea(info)));
#ifndef HANS
   XClearWindow(Ixm_display, XtWindow(GetWindowInfoDrawingArea(info)));
#endif
   fprintf(stderr, "2\n");
   RecursiveDrawElement(GetWindowInfoMainGroup(info), False);
   /*
   DrawSelected(info);
   */
   DrawCorners(info);
}
```

Chapter 7 Bibliography

[1] *Neural Networks; A Comprehensive Foundation*, Simon Haykin, Macmillan College Publishing Company Inc., First Edition, 1994.

[2] *Knowledge–Based Artifical Neural Networks*, Geoffrey G. Towell amd Jude W. Shavlik, Artificial Intelligence, Volume 69 or 70, 1994.

[3] A Hybrid Intelligent Architecture and Its Application to Water Eservoir Control, Ismail Taha and Joydeep Ghosh, Department of Electrical and Computer Engineering, 1995.

[4] *InterAct*, Neuro Technology GmbH, Version 4.4, 1993–1995, UNIX / X–Windows.

[5] *NeuralNet*, Eugene W. Hodges, Version 2.0, 1992, UNIX / X–Windows.

[6] *NeuroSolutions*, NeuroDimension Inc., Version 3.0b6, 1994–1997, DOS / Windows.

[7] *NCS NEUframe*, Neural Computer Sciences, Version 3,0,0,0, 1997, DOS / Windows95 4.0.

[8] *SNNS*, Institute for Parallel and Distributed High Performance Systems, University of Stuttgart, Version 4.0, 1995, UNIX/X–Windows.

[9] *Interleaf 6*, Interleaf Inc., Version 6.0.1, 1993, UNIX/X–Windows and DOS/Windows.

[10] *Common Desktop Environment: Style Guide and Certification Checklist*, Open Software Foundation Inc., First Edition, 1989, 1990, 1993, 1994.

[11] *Fuzzy logic with engineering applications*, Timothy J. Ross, Mc. Graw–Hill Inc., First Edition, 1995

[12] *X Window System Programming*, Nabajyoti Barkakati, Sams Publishing, Second Edition, 1994.

[13] *The X window System; Programming and applications with Xt*, Douglas A. Young, PTR Prentice Hall, Second Edition, 1994.